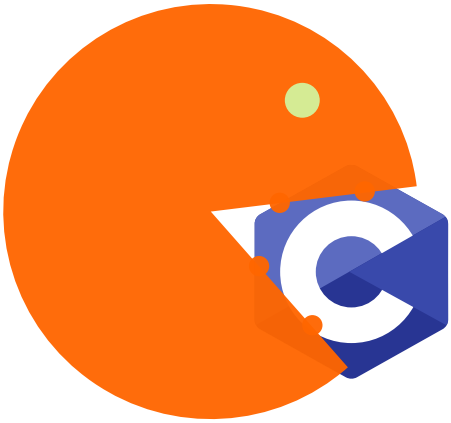


초보자를 위한 C언어 강의

씹어먹는 C언어

이재범

v.2020.06.20

modoocode.com

Copyright © 2019-2020 이재범

이 책은 **모두의 코드**에 연재된 씹어먹는 C 언어 강좌를 책으로 옮긴 것입니다. 해당 강좌는 https://modoocode.com 에서 볼 수 있습니다.

차 례

차 례 **3**

제 1 장 C 언어의 세계로 **1**
C 언어 본격 맛보기 14
주석(Comment) 19
컴퓨터에서 수를 표현하는 방법 22

제 2 장 변수 (variable) **30**

제 3 장 계산 하기 **40**
컴퓨터가 음수를 표현하는 방법 (2의 보수) 55

제 4 장 문자 입력 받기 **61**

제 5 장 만약에... (조건문) **69**

제 6 장 반복문 (for, while) **89**
리눅스에서 C 프로그래밍 하기 108

제 7 장 switch 문 **109**

제 8 장 형 변환 (타입 캐스팅) **118**

제 9 장 C 언어의 아파트 - 배열 **128**
다차원 배열 148

제 10 장 포인터 **159**
상수 포인터, 포인터의 덧셈 뺄셈, 배열과 포인터 171
포인터의 포인터 190

제 11 장 함수 (function) **210**
포인터로 받는 인자 224
여러가지 인자들 241
생각해볼 문제에 대한 아이디어 253

제 12 장 디버깅(Debugging) **261**

제 13 장 문자열 (string) **269**
버퍼에 대한 이해 280
문자열 리터럴에 대한 이해 294
도서 관리 프로젝트 309

제 14 장 구조체 (struct) **334**
구조체를 인자로 받는 함수 347
공용체(union)와 열거형(enum) 361

제 15 장 변수의 생존 조건과 데이터 세그먼트의 구조 **375**

제 16 장 여러 파일로 나누기 **388**
여러 전처리기 구문과 라이브러리 사용법 409

제 17 장 void 타입과 main 함수에 대한 이해 **420**

제 18 장 동적 메모리 할당 (dynamic memory allocation) **434**
구조체의 동적 할당과 메모리 관리 함수 447

제 19 장 매크로 함수, 인라인 함수 **461**

제 20 장 그 밖에 키워드들 **472**

제 21 장 파일 입출력 **487**
파일 안에서 이동하기 499
도서 관리 프로그램의 완성 525

제 22 장 C 코드 최적화 **536**

1

C 언어의 세계로

안녕하세요 여러분! 씹어먹는 C 언어 강좌에 오신 것을 환영합니다.

왜 C 언어를 배워야 하는가?

아마도 이 강좌를 보고 있는 여러분들은 분명 누군가에 의해 "너는 C 언어를 배워야만 해!" 해서 들어 오셨을 것입니다. 하지만 많은 경우 도대체 왜 굳이 C 언어를 배워야 하는지 알려주지는 않았을 것입니다. 저 또한 C 언어를 처음 공부한 이유가 누가 배워야 한다고 해서 배운 것이지, 제가 필요성을 느껴서 배운 것도 아닙니다.

여러분이 프로그래밍을 시작한 계기는 분명 여러가지가 있을 것입니다. 저는 처음 프로그래밍을 배운 계기가 게임을 만들고 싶어서였습니다. 어떤 분들은 멋진 웹사이트를 만들고 싶었을 것이고, 또 어떤 분들은 안드로이드 앱을 만들고 싶어서 였을 것입니다. 그런데 이러한 것들을 하기 위해 굳이 C 언어를 알아야 할까요?

사실 아닙니다. 요즘에 게임을 만드는 가장 핫한 플랫폼인 유니티는 보통 C# 으로 프로그래밍 하고, 웹사이트의 경우 대개 파이썬, PHP, 자바스크립트 등등으로 만듭니다 (여기서 말하는 웹사이트는 백엔드). 또 안드로이드 앱은 자바 (혹은 코틀린), iOS 앱은 스위프트나 Objective-C 로 만듭니다. 여기 어디에도 C 는 쓰지 않습니다. **아마 이러한 것들을 지금 빨리 만들고 싶어서 손이 근질근질 거리시는 분들은 이 부분에서 뒤로가기를 누르셔도 좋습니다.**

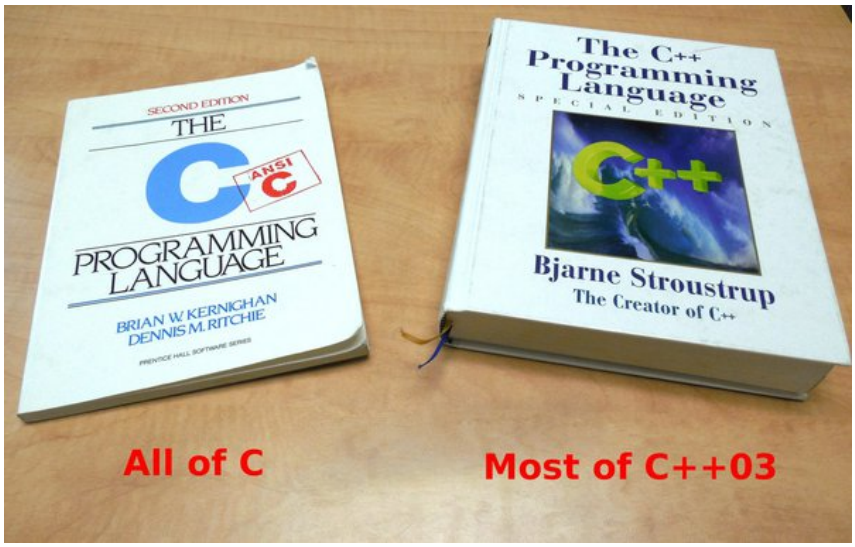
그렇다면 왜 사람들은 C 언어를 배우라고 할까요? 도대체 왜 학교에선 다른 언어들 놔두고 C 언어를 가르칠까요? 제가 생각하는 이유는 다음과 같습니다.

- 만약 적당히 잘하는 프로그래머가 목표라면, 굳이 컴퓨터 내부가 어떻게 돌아가는지 몰라도 괜찮습니다. 하지만 좋은 프로그래머가 되려면, **컴퓨터의 내부 원리를 아는 것이 필수적**입니다.
  만약에 C 언어를 배우게 된다면, 컴퓨터 내부 원리를 더 쉽게 이해할 수 있습니다.

- C 언어를 배운다면, 다른 언어를 더 쉽게 습득할 수 있습니다. 많은 언어들이 C 언어에서

파생되어서 생겨났습니다 (위에서 나열한 C#, Objective-C, 그리고 가장 유명한 C++ 까지 모두 이름에서 알 수 있듯이 C 의 영향을 많이 받은 언어들 입니다.) 따라서 이런 언어들을 배우는데 많은 도움이 됩니다.

- 이미 엄청나게 많은 코드들이 C 언어로 작성되어 있습니다. 따라서, 어느 정도 수준 이상에 도달하게 된다면 C 언어를 결코 피하실 수 없을 것입니다.



참고로 제일 최근에 나온 C++ 은 C++ 20 으로 그 내용을 다 포함한다면 저 책의 2 배는 되야 할 것입니다.

- 마지막으로, C 언어는 배워야 할 내용이 다른 언어에 비해 매우 적습니다! (C 언어의 최고의 교과서라 불리는 *The C Programming Language* 라는 책은 200여 페이지 밖에 되지 않습니다. 반면에 C++ 에서 비슷한 역할을 하는 책인 *The C++ Programming Language* 는 1300페이지가 넘습니다.) 물론, **배워야 할 내용이 적다** 와 **쉽다** 는 차이가 있으니 유의해두시기 바랍니다 :)

이러한 연유에서 저는 여러분들이 C 언어를 배워야 하겠다라고 마음 먹기 매우 잘한 일이라 생각합니다. 특히 C 언어를 배우는데 있어서 제 강좌인 씹어먹은 C 언어를 선택한 것 역시 훌륭한 선택이라 자부합니다.

어떤 식으로 배워야 할까요?

흔히 언어를 가장 쉽게 배우는 방법은 그 말을 하는 친구를 사귀는 것이라 하였습니다. (여자/남자 친구면 더 좋구요!) 이런 식으로 배우는 것이 딱딱한 문법책 외우면서 단어만 죽어라 외우는 것 보다 더 재밌고 실력도 금방 금방 늡니다.

C 언어도 마찬가지 입니다. 더군다나 우리는 C 언어를 매우 잘하는 친구를 이미 사귀고 있습니다. 바로 여러분들이 지금 사용하고 있는 컴퓨터가 있잖아요. C 언어를 가장 빨리 습득하는 방법은 강좌에서 배운 내용을 토대로 간단한 프로그램을 직접 만들어 보는 것입니다. 여기에 재미를 붙인다면 더 빠르게 배울 수 있겠지요!

저는 누구인가요?

저는 C 언어를 전문적으로 가르치는 강사는 아닙니다. 전문적인 강의를 원했던 사람은 여기서 다시 살포시 뒤로가기를 눌러도 됩니다. (아마 80% 이상이 누를 것이라 추정) 하지만 제가 유일하게 잘 할 수 있다고 자부하는 것은 어려워만 보이던 C 언어의 기초 부분을 최대한 쉽게 설명하는 것입니다.(저에 대한 자세한 프로필은 여기를 참조해주세요)

앞으로 저와 함께 C 언어의 세계로 여정을 떠날 사람들은 준비가 되었습니까? 그럼 모두 스크롤을 내려주세요!

여정에 필요한 준비물

C 언어를 배우기 위해선 다음과 같은 준비물이 필요합니다.

1. 컴퓨터
2. 머리
3. 노오력
4. 컴파일러.

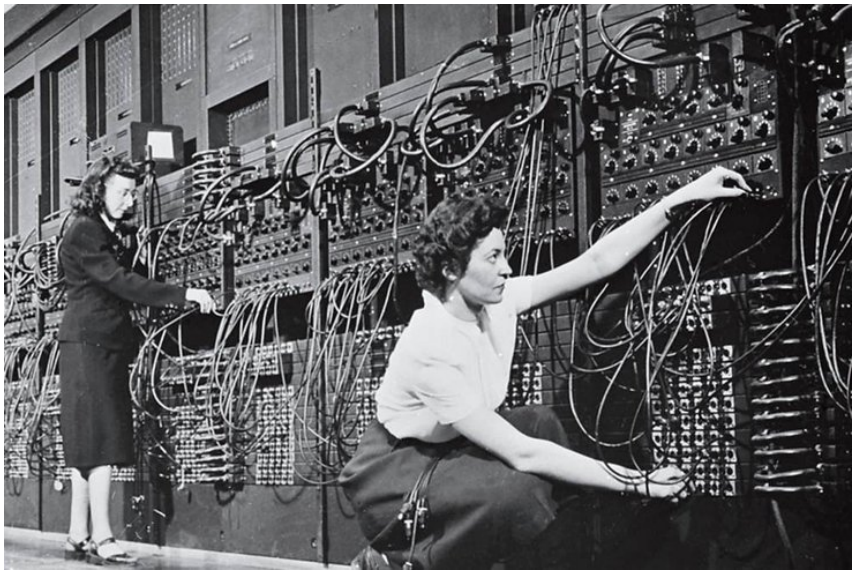
1,2,3 번은 여러분들 께서 충분히 준비할 수 있다고 생각합니다.

그런데 4번, 컴파일러는 뭐지?

컴파일러?

컴파일러, 영어로는 *Compiler* 라고 씁니다. 말그대로 컴파일(Compile) 해주는 것(-r) 입니다. 그렇다면 컴파일은 도대체 무엇일까요?

아마 여러분은 컴퓨터는 0 과 1 밖에 모르는 바보(?) 라고 들었을 것입니다. 맞습니다. 컴퓨터의 두뇌라고 할 수 있는 **CPU** 에서는 수 많은 0 과 1 들이 이리저리 돌아다니고 있을 것입니다. 만약에 이런 컴퓨터에 명령을 내리려면 오직 0 과 1 로만 써야겠지요? 010101010111 이런식으로 말입니다.



C 언어는 노오력으로 배울 수 있지만, 이건 아마 노오오오오오력이 필요햇을 것입니다!

아마 옛날에는 실제로 전선을 이리저리 연결해가며 비슷한 방식으로 명령을 내렸을 것입니다.

만약 우리가 프로그래밍을 0 과 1 로 만 한다면 얼마나 끔찍할지 생각해보세요. 만약 이런 세상이였다면 프로그래머 연봉이 평균 10억은 되었을 것입니다. 하지만 훌륭한 컴퓨터 과학자들 덕분에, 0 과 1 대신에, 그나마 사람이 알아들을 수 있는 '언어'로 프로그래밍을 할 수 있게 하였습니다. 0 과 1 로 표현되는 명령을, 사람들이 그나마 알아듣기 쉽게 표현하게 말입니다. 예를 들어서 1 + 1 을 하기 위한 명령을 컴퓨터에 내린다면

```
01011101010101
```

이 되겠지만 컴퓨터 언어를 통해 간단히

```
1 + 1
```

이렇게 표현할 수 있을 것입니다. 문제는 컴퓨터가 인간의 언어를 도무지 알아 들을 수 없으니, 컴퓨터가 이해할 수 있는 0..1 들로 바꿔주는 녀석이 필요합니다. 이렇게, 사람들이 사용하는 '프로그래밍 언어'와 컴퓨터가 이해하는 '기계어' 사이 다리 역할을 수행하는 것이 바로 **컴파일러** 입니다.

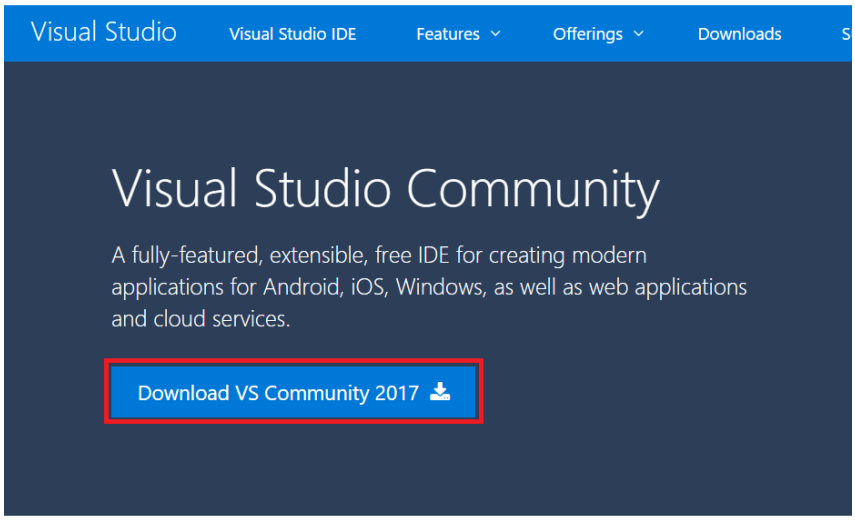
좋은 소식은 쉽게 컴파일러들을 구할 수 있다는 점입니다. 유명한 컴파일러로는 무료로 배포되는 **gcc** 와 **clang** 를 들 수 있지만 사용법이 초보자들에게는 약간 복잡합니다. 대신에, 저희는 마이크로소프트에서 무료로 배포하는 **Visual C++ 2017 Community** 를 사용할 것입니다.

더 좋은 소식은 Visual Studio Community 2017 을 다운받으면 아예 위 컴파일러 뿐만이 아니라 사용자들이 프로그래밍 하기 편하게 여러가지 프로그램들이 딸려 옵니다. 땡큐죠!

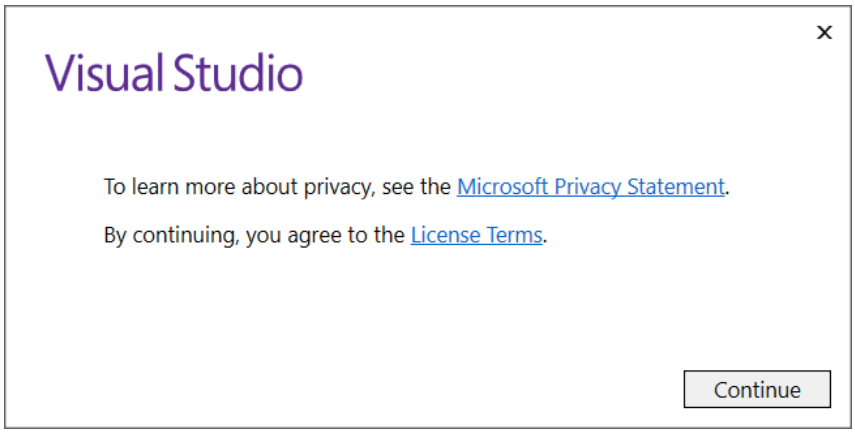
비주얼 스튜디오 커뮤니티 2017 은

https://www.visualstudio.com/vs/community/

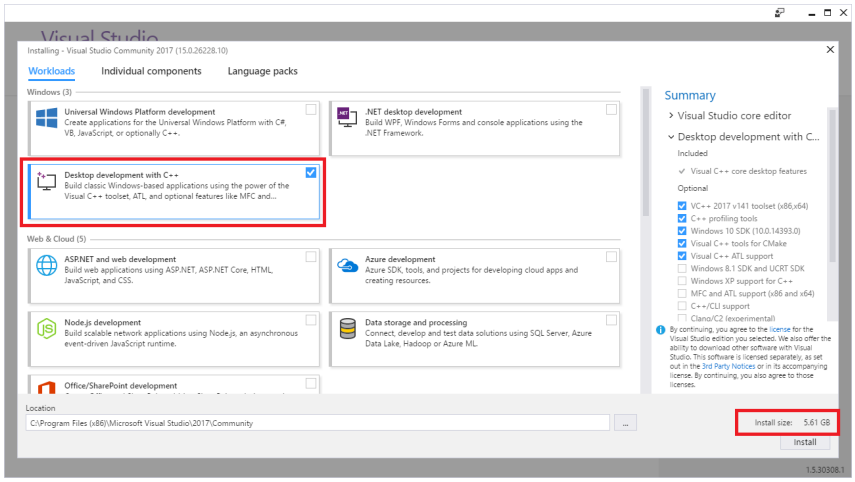
에서 받으실 수 있습니다.



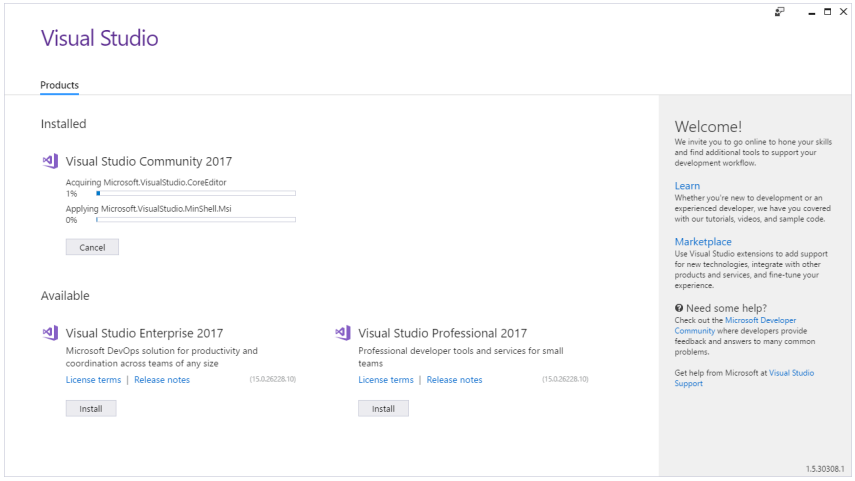
위와 같이 사이트에 들어가서 *Download VS Community 2017* 을 누르시면 됩니다. (제가 지금 미국에서 있어서 영어로 뜨지만 아마 한국에 계신 분들은 한국말로 뜰 것입니다 :)



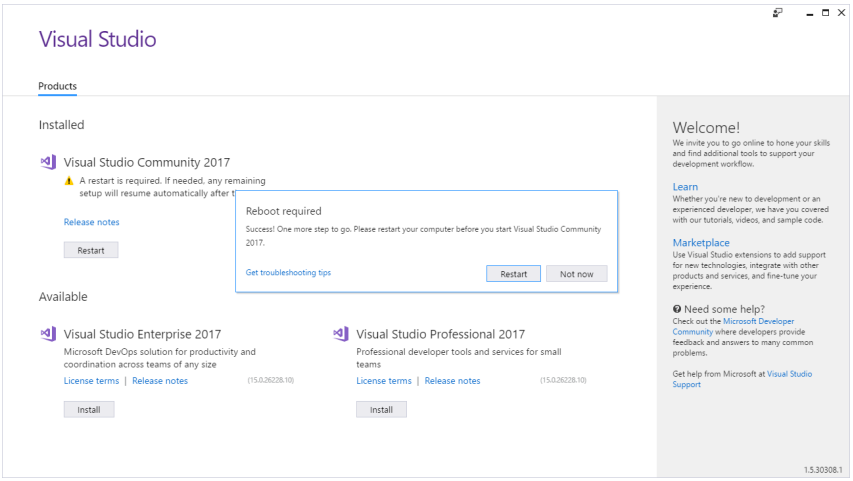
다운로드 된 실행파일을 실행하면 위와 같이 뜹니다. *Continue* 를 눌러주세요



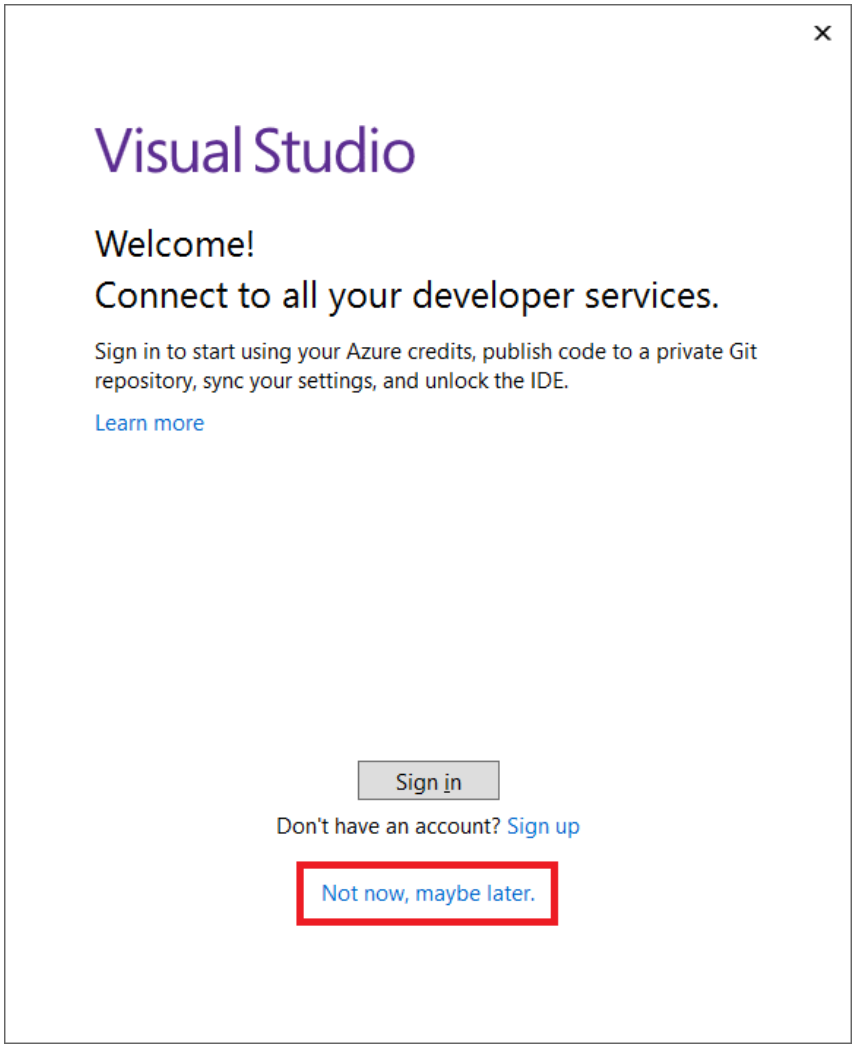
이제 어떠한 것들을 설치할 지 선택할 수 있는데, 일단 여러분은 C/C++ 을 배우는 것이 목적이므로 화면에 **Desktop development with C++** 만 선택해 주시면 됩니다. 혹시 내가 나중에 C# 이나 기타 등등으로 더 많은 코딩을 하고 싶다 그러시는 분들은 다른 것들을 선택해서 설치해주셔도 상관 없지만, 설치 용량이 엄청 커지게 됩니다. 저거 하나만 선택했는데에도 벌써 5.61 GB 네요.



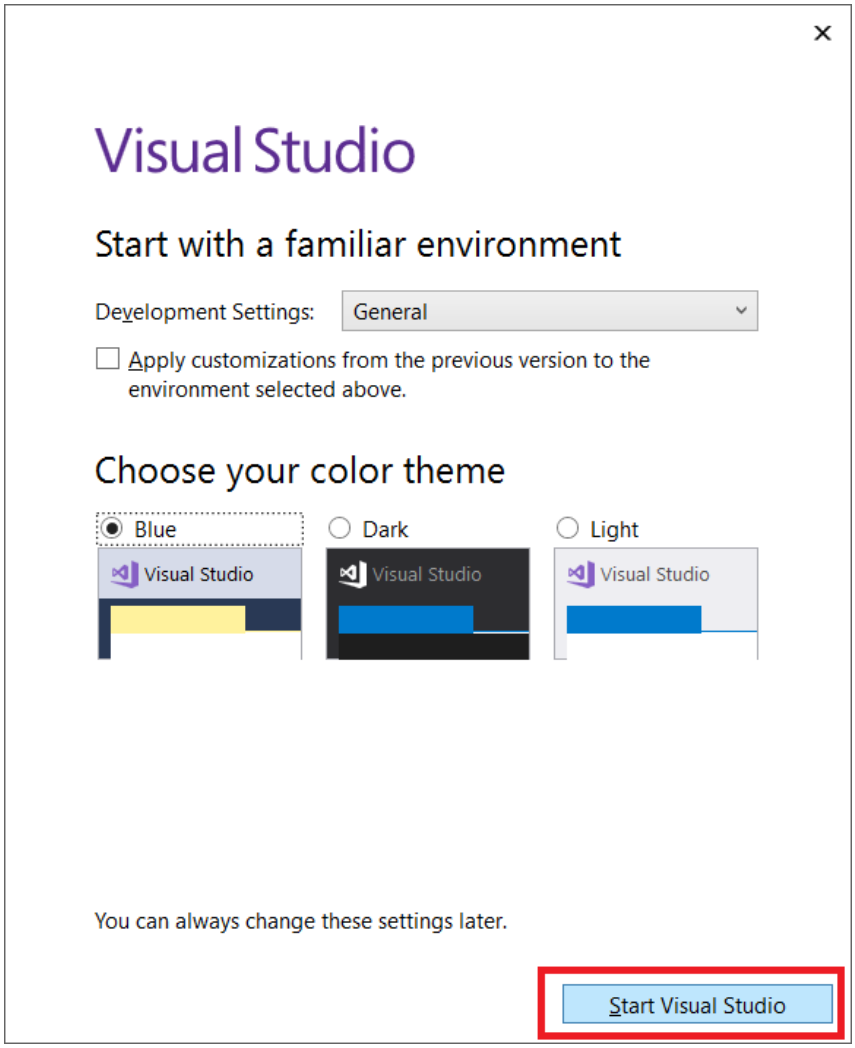
그럼 이제 알아서 필요한 것들을 다운 받아서 설치하게 됩니다. 인터넷 상황에서 따라서 30분에서 1시간 정도 기다려야 합니다.



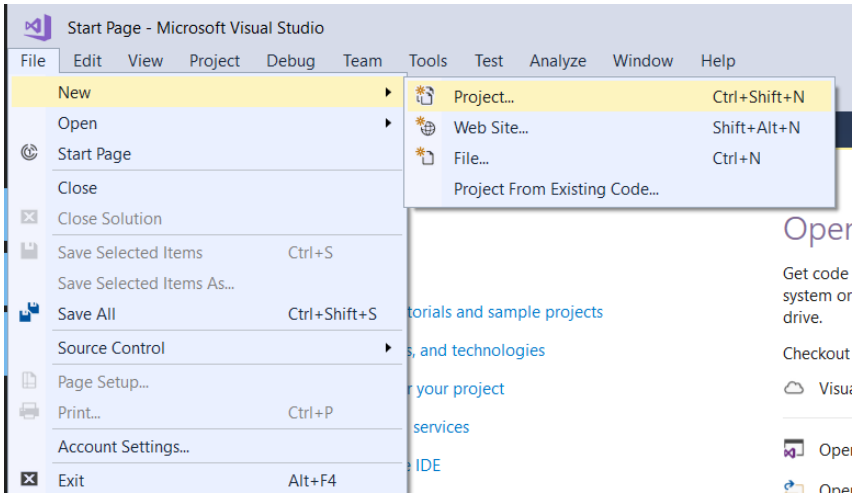
설치가 다 되었으면 컴퓨터를 재시작 해야 합니다.



뭔가 가입하라고 나오는데 그냥 나중에 한다고 하고 무시하면 됩니다.

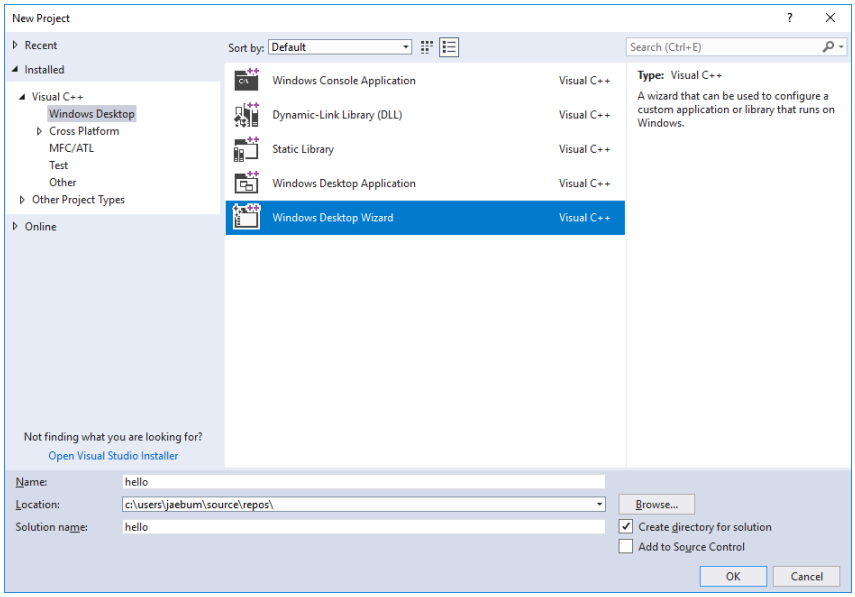


그 다음으로 어떤 테마를 고를지 정하면 됩니다. 저의 경우 그냥 디폴트인 파란색 바탕을 사용하는데, 사람들에 따라서 어두운 테마를 좋아하는 경우도 있습니다 :)

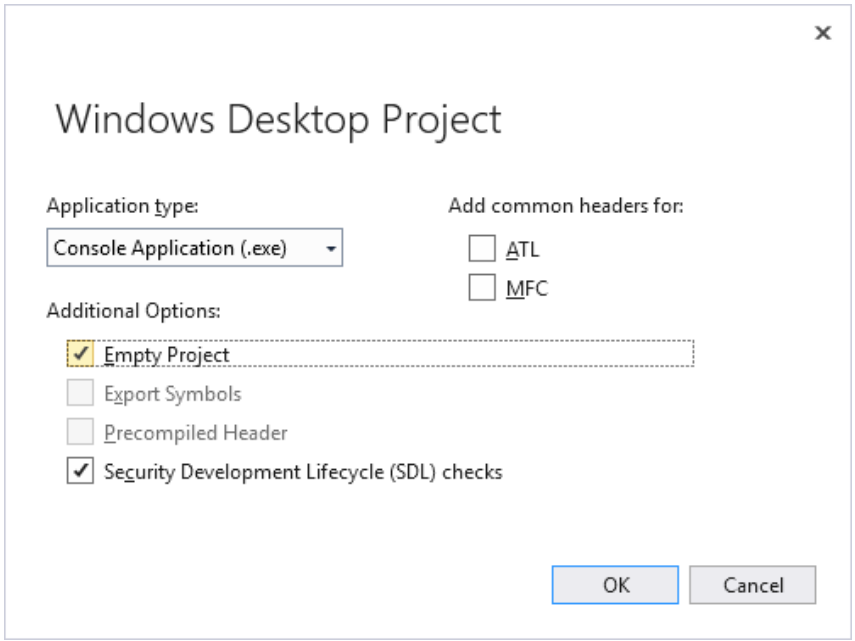


이제 메인 화면에서 새로운 프로젝트를 만들어봅시다.

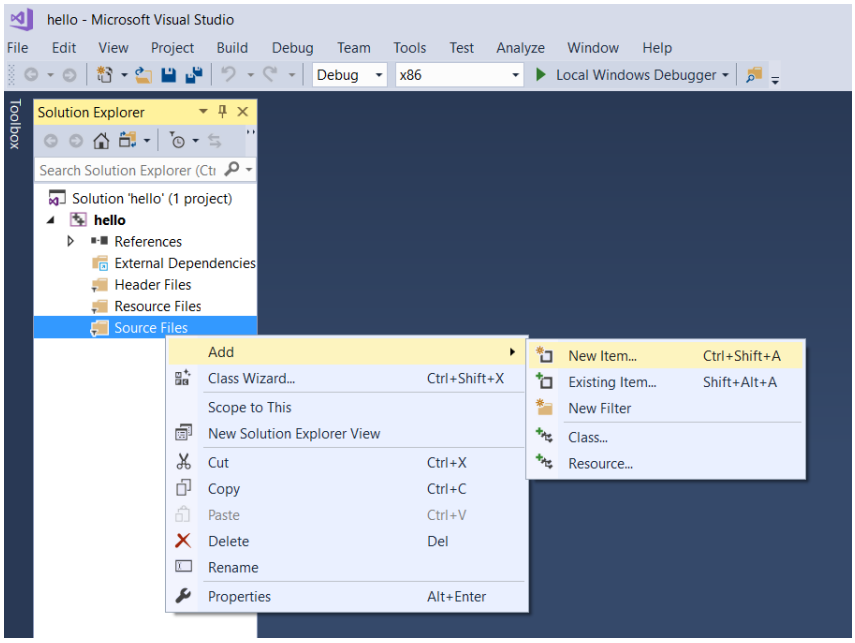
간단히 `Ctrl + Shift + N` 을 누르면 새 프로젝트를 만드는 창을 띄울 수 있습니다.



다음으로 프로젝트 선택 화면에서 **Windows Desktop Wizard (Windows 데스크톱 마법사)** 를 선택합니다.

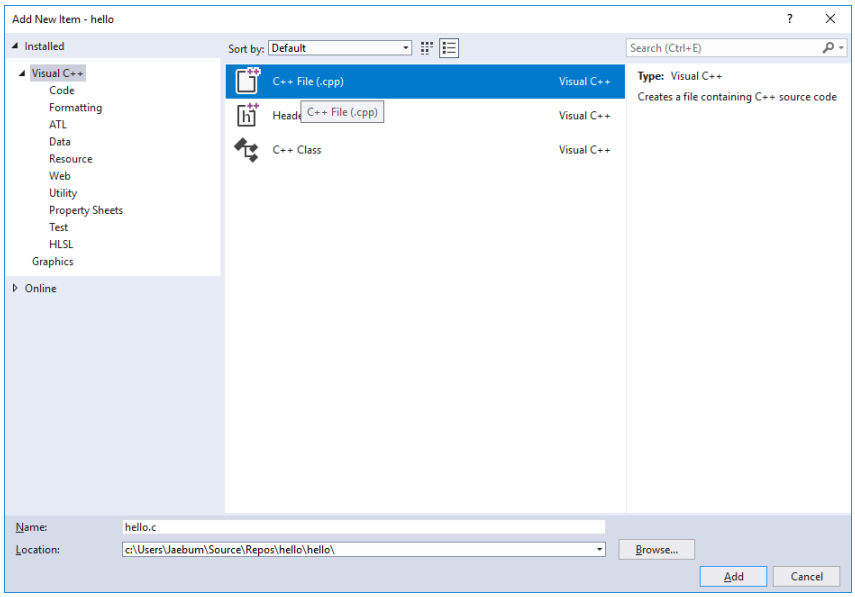


그 다음, 위와 같은 화면에서 Application type 에 **콘솔 프로그램 (Console Application)** 을 선택한 후, 아래에 **빈 프로젝트 (Empty Project)** 에 체크해줍니다. 그 후 확인을 눌러주면 프로젝트가 생성됩니다.



이제 프로젝트에 소스 파일을 추가해야 합니다.

기본적으로 왼쪽에 보시면 솔루션 탐색기에서 소스 파일을 쉽게 추가할 수 있는데, 솔루션 탐색기가 안보인다면, `Ctrl + Alt + L` 을 눌러서 띄울 수 있습니다.



위 화면에서 C++ 파일을 클릭한 뒤에, 이름에 `hello.c` 라고 적습니다.[1)]

그럼 이제 왼쪽에 만들어진 hello.c 파일을 클릭합니다.

1) C++ 파일을 선택하였지만, 파일 확장자가 .c 이므로 비주얼 스튜디오에서 알아서 C 컴파일러로 컴파일 해줍니다.

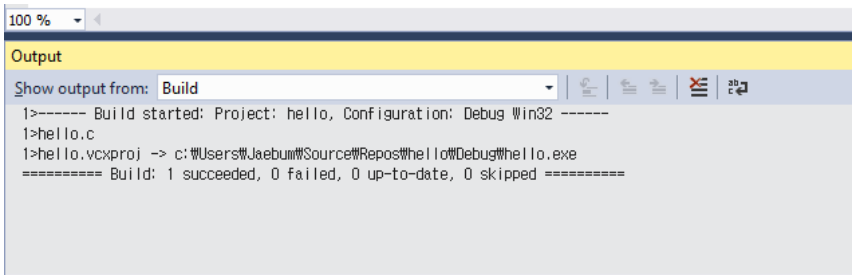
내 생애 첫 프로그램

드디어 여러분 인생 첫 프로그램을 만들 시간이 다가왔습니다. 에디터 화면에 다음과 같이 입력합니다.

```c
#include <stdio.h>
int main() {
  printf("Hello, World! \n");
  return 0;
}
```

한 가지 당부 드리고 싶은 말은, 반드시 손으로 직접 입력해 보기 바랍니다. 그냥 `Ctrl - C, Ctrl - V` 하는 것은 시간은 절약되지만 결국 나중에 머리에 남는 것은 없게 됩니다.

이제, 위 내용을 다 입력하였으면 `F7` 를 눌러 주어서, 또는 상단의 빌드 → 솔루션 빌드를 눌러서 컴파일 합니다. 만약 위 내용을 잘 써서 성공적으로 빌드 되었다면 아래 아래와 같은 화면을 보게 될 것입니다.



성공적으로 컴파일을 하였습니다

그런데, 간혹 가다 어떤 사람들은 오류가 뜨는 사람들도 있는데, 대표적으로

컴파일 오류

```
error C2143: 구문 오류 : ';'이(가) 'return' 앞에 없습니다
```

라 던지,

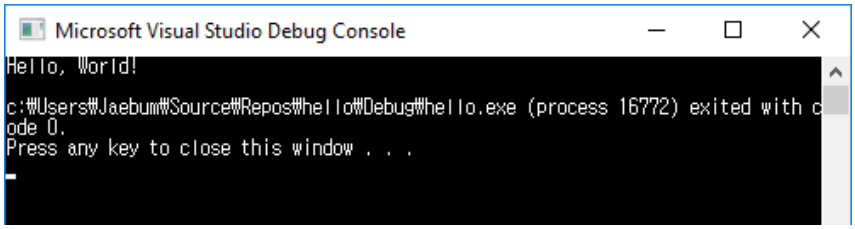
컴파일 오류

```
error C2001: 상수에 줄 바꿈 문자가 있습니다.
error C2143: 구문 오류 : ')'이(가) 'return' 앞에 없습니다.
```

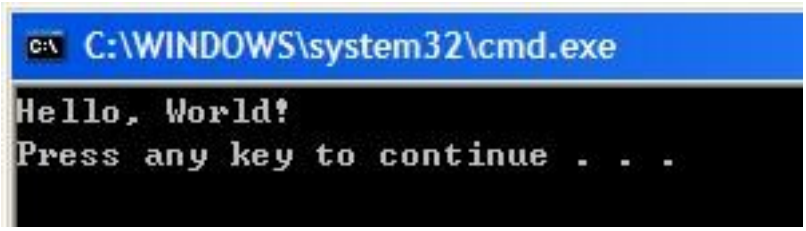
와 같은 오류를 보게 되는 사람들이 있습니다.

첫 번째의 경우, `printf("Hello, World! \n")` 다음에 세미 콜론 (`;`) 을 쓰지 않아서 나타나는 경우이고, 두 번째의 오류는 `printf` 안에 큰 따옴표로 제대로 닫지 않았을 경우입니다. 위에서 언급한 오류가 아니더라도, 어디선가 오류가 발생하였따면 십중 팔구 위의 소스 코드를 잘못 쳤기 때문 이므로 다시 한 번 신중히 쳐보거나, 그래도 안되면 복사해 보시기 바랍니다.

이제, 드디어 이렇게 완성된 프로그램을 실행할 시간이 다가왔습니다. `Ctrl + F5` 를 눌러서 프로그램을 실행해 봅시다.



Hello, World! 가 출력되었습니다. 밑에 나온 자질구레한 메세지는 무시해도 됩니다.



2008년 윈도우 XP 쓰던 시절의 출력 화면

만세! 위와 같이 Hello, World! 가 성공적으로 출력됨을 확인할 수 있습니다. 아래 나오는 자질구레한 메세지는 무시해도 됩니다. 그 부분은 여러분이 쓴 코드 때문에 나온 것이 아니라 비주얼 스튜디오 자체에서 추가한 메세지 입니다.

참고로 아래 사진은 2008년 강좌를 처음 제작했을 때 나온 프로그램 결과 입니다. 10년 이 넘게 흘렀는데, 모습은 비슷하지요?

아무튼 방금 여러분은 인생 첫 프로그램을 만들게 되었습니다!

다음 강좌에서는, 위 프로그램이 도대체 어떻게 동작하는 것인지에 대해 알아보도록 합시다.

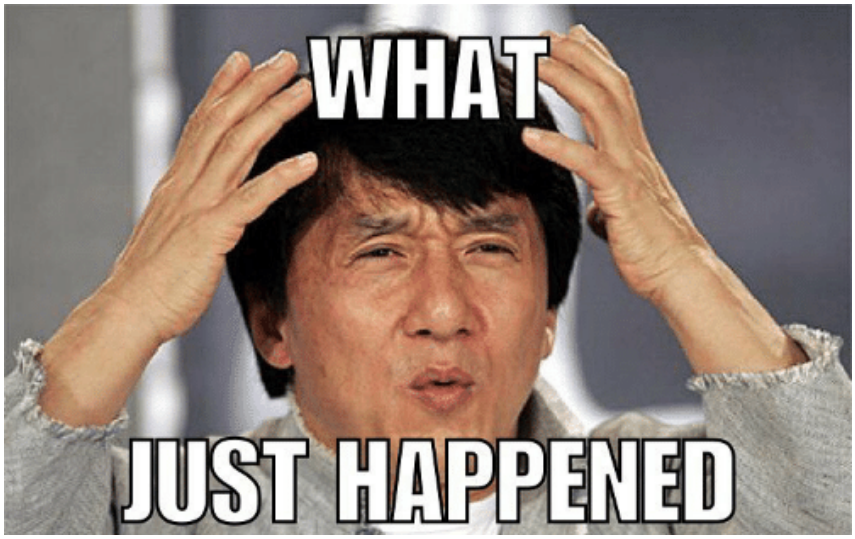
뭘 배웠지?

- C 언어는 (요새 나오는 언어들에 비해) 조금 어려워보일 수 있지만, C 언어를 배움으로써 얻을 수 있는 것들이 더 많습니다! 여러분은 훌륭한 선택을 하신 것입니다!
- 컴퓨터는 0 과 1 밖에 모르기 때문에 컴퓨터에게 명령을 내리기 위해서는 **컴파일** 이라는 과정을 통해서 사용자가 입력한 코드를 변환해야 합니다. 이를 수행하는 프로그램을 **컴파일러** 라고 합니다.
- 비주얼 스튜디오를 설치하였고, 해당 IDE 의 컴파일러로 코드를 컴파일 하였습니다.
- 여러분 인생 첫 프로그램을 작성하였습니다.

C 언어 본격 맛보기

안녕하세요 여러분. 저번 강의에서의 희열을 아직도 느끼시나요? 방금 자신의 손으로 최초의 프로그램 - `Hello, World!` 를 만들었다는 사실을 말이죠. 하지만 자신이 프로그램을 만들었다는 사실을 친구들에게 자랑하기 전에, 그 프로그램이 어떻게 동작하는지 살펴보도록 합시다.

도대체 뭔일이 있던거지?



Hello, World 프로그램 분석하기

지난번 강좌에서 **도대체 내가 뭘 쓰고 있는 걸까** 라고 생각하면서 코드를 짜셨을 것입니다.

그래도 한 가지 눈치 챘을 법한 부분은 바로 큰 따옴표로 닫혀 있는 부분의 "Hello, World!" 가 프로그램에 출력된다는 점입니다. 한 번 다른 문장으로 바꿔서 과연 그 문장이 출력되는지 확인해보는 것도 좋습니다.

그렇다면 우리가 작성한 코드가 어떠한 의미를 가지는지 살펴보도록 하겠습니다.

```c
#include <stdio.h>
int main() {
  printf("Hello, World! \n");
  return 0;
}
```

일단 위 프로그램의 첫 줄 부터 봅시다.

```
#include <stdio.h>
```

영어를 잘 하시는 분은 include 의 뜻이 '포함하다' 라는 것임을 알 수 있습니다. 그렇다면 위 프로그램은 무엇을 포함하고자 하는 것일까요? 바로 옆의 stdio.h 라는 파일을 포함하고자 하는 것입니다.

우리는 왜, stdio.h 라는 파일을 이 프로그램에 포함 시켰을 까요? 그 이유는 아래에서 설명하도록 하겠습니다.

그 다음 부분을 살펴 봅시다.

```
int main()
```

이번에는 조금 생소한 단어군요. main 은 그렇다 쳐도, int 는 또 뭘까요? RPG 게임을 많이 하신 분들이라면 지능을 뜻하는 *intelligence* 의 약자라고 생각하실 수 도 있지만, 사실 이는 **정수** 를 뜻하는 **integer** 의 약자입니다. 또한 그 옆의 main 은 함수를 말하는 것이죠.

사실 이 문장의 뜻은 '정수 형을 반환하는 메인 함수' 라는 뜻이며, 모든 C 프로그램은 이 main 에서부터 시작됩니다. 자세한 사실은 나중에 알아봅시다.

```
{
```

그 다음 문장은 참으로 간단하네요. 중괄호 입니다. 여기서 중괄호는 main 함수의 시작을 알리게 됩니다. 즉, 중괄호로 묶인 부분은 '여기는 main 함수 꺼야' 라는 것을 나타냅니다.

C 언어에서는 main 함수 뿐만이 아니라 어떠한 문장도 여는 중괄호가 있다면 반드시 이에 대응되는 닫는 중괄호(}가 와야 합니다. 여기서도 마찬가지로, 마지막 문장에서 닫는 중괄호가 와있네요.

```
printf("Hello, World! \n");
```

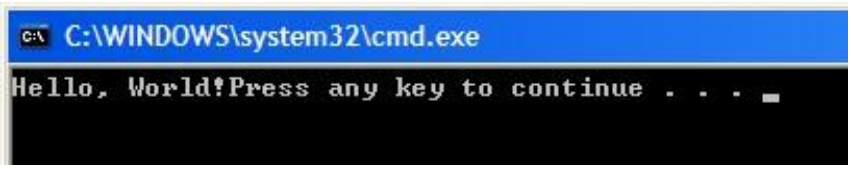
이제, 위 프로그램에서 가장 핵심이라 볼 수 있는 부분인 printf 를 살펴 봅시다. printf 는 화면에 괄호안의 내용을 출력할 수 있게 해주는 함수 입니다. 위의 경우, 괄호 안에 있는 Hello, World! 가 화면에 출력되었습니다.

그런데, 도대체 위 함수가 어떻게 해서 화면에 글자를 출력하는 것일까요? 사실, 화면에 글자를 출력하는 것은 쉬운 일 일 것 같지만, 매우 복잡한 과정을 거치는 것입니다. 왜냐하면, 일단 운영체제에 자신이 화면에 글자를 뿌려야 한다는 메시지를 보내야 하고, 또 운영체제는 하드웨어 (모니터) 에 이를 뿌린다는(출력한다는) 것을 이야기 해 주어야 하기 때문이죠.

하지만 우리가 위 짧은 문장을 화면에 표현하기 위해 위 모든 내용을 작성해야 한다는 것은 상당히 불합리해 보입니다. 따라서 우리는 위 모든 내용을 포함하고 있는 파일을 필요로 하는데, 그 것이 바로 앞서 이야기한 stdio.h 입니다. **(studio 가 아닙니다!)**

`stdio` 는 *STandard Input Output header* 의 약자로, **표준 입출력 헤더** 입니다. 이 파일에는 입출력, 즉 화면에 출력하고, 키보드로 부터 입력을 받아들이는 것을 담당하고 있습니다. 물론, 이 파일 하나에 모든 내용이 다 구현 되어 있는 것은 아닙니다. 자세한 내용은 나중에 배우게 됩니다.

그런데, 한 가지 이상한 점이 있습니다. 큰 따옴표 안의 내용이 모두 출력되는데, 왜 마지막의 `\n` 은 출력되지 않은 것일까요? 그렇다면 한 번 여러분들 께서 `\n` 을 지워 보고 다시 프로그램을 실행해 보세요. 아마 다음과 같이 나올 것 입니다.



지난번 하고 차이점이 보이세요? 분명히 지난 번에는 *Press anykey to continue* 가 한 줄 개행되어 나타났는데 이번에는 연이어 나타났습니다. (윈도우 한글판 사용자의 경우 '아무키나 누르세요' 가 나타날 것입니다)

아하, 알겠습니다. 바로 `\n` 은 키보드 상의 엔터, 즉 개행 문자 였던 것입니다. (참고로 `\` 를 `Escape character` 라고 합니다)

참고적으로 알아야 할 사실은 우리나라 키보드의 경우 `\` 로 나타나지만 외국 대부분의 키보드에는 `\` 대신에 역슬래시(＼) 를 사용합니다. 따라서, 보통 C 언어 서적을 보면 ＼n 이라 나타난 것이 있는데 이는 `\n` 과 똑같은 것입니다.

마지막으로 중요한 점은, 모든 문장이 끝나는 부분에 세미콜론(;)을 찍어 주어야 된다는 것입니다.

물론, 함수의 선언 부분 (즉, `int main()`) 뒤에나 헤더파일 선언 부분 (`#include <stdio.h>`) 뒤에는 `;` 을 붙이면 안되지만, 위와 같이 `printf(.....)` 나, 아래 줄의 `return 0` 와 같은 문장들에게는 꼭 끝에 세미콜론을 붙여아합니다. 만약 붙이지 않는다면 이전 강의에서 보았던 오류들이 나타나게 됩니다.

```
return 0;
```

영어로 읽어 보면 대충 뜻을 짐작하셨겠지만, 0 을 **반환(return)**한다는 뜻 입니다. 0 을 왜 반환할까요? 그리고 그 것을 반환한다면 '누구' 한테 반환하는 것인가요? 쉽게 말해 운영체제에게로 반환합니다. (정확히 말하면 이 프로그램을 호출한 프로그램) 그런데 왜 하필이면 0 일까요? 1 이면 안되고 왜 2 이면 안되죠.

그렇다면 한 번 1 이나 다른 원하는 숫자를 반환하도록 해보세요. 결과는 똑같습니다. 그런데 왜 굳이 0 을 반환하는 것일까요?

사실은 0 을 반환한다는 것은 컴퓨터에게 프로그램이 무사히 종료되었음 을 알리는 것이죠. 반면에 1 을 반환한다면 컴퓨터에게 프로그램이 무사히 종료되지 않았어요 - 오류가 발생했어요. 를 알리는

것입니다.[2)]

```
}
```

마지막으로 이렇게 꼭 중괄호로 닫아주어야지, 그렇지 않을 경우 파일의 끝이 없다는 오류가 발생하게 됩니다. 와우! 이쯤 되면 위 프로그램을 빠삭하게 분석해 보았다고 할 수 있습니다.

주석(Comment) 넣기

마지막으로 모든 프로그래밍 언어에 기본으로 있는 기능이자, 그 만큼 중요한 기능인 주석 넣기에 대해 알아봅시다.

주석이라 하면, 코멘트, 즉 **자신의 코드에 대해 설명을 해주는 것**입니다. 아마 위의 대여섯 줄 짜리 코드에 뭐가 설명할 필요가 있겠어? 라고 생각할 수 있지만 실제로 '쓸만한' 프로그램을 만들게 되면 코드의 길이가 수천줄을 넘어가는 것은 예삿일입니다. 물론 그런 파일들이 여러개 모여서 프로그램을 만들게 되는 것이지요.

그렇게 된다면 코멘트 없이는 이 코드가 도대체 무슨 역할을 하는지도 잘 모르고, 남이 쓴 코드가 어떤일을 하고, 어떻게 돌아가는지 이해가 잘 안되는 경우가 많습니다. 더욱 심각한 사실은 자신이 쓴 코드도 못알아 보는 경우가 있다는 것입니다. 이처럼, 이런 기분나쁜 일을 미연에 방지하기 위해 이 코드가 무슨 역할을 하고 어떻게 동작되는지 간단하게 나마 설명해 주는 것이 필요하겠죠. 그런 것을 바로 주석 이라 합니다.

우리가 코드를 이해하기 위해 필요한 것이지, 컴퓨터에는 아무런 도움이 되지 않는 것이므로 컴파일러는 이 주석을 완전히 무시해 버립니다. 마치 우리만이 볼 수 있는 것 처럼 말이죠. 그렇기 때문에 주석에 무슨 짓을 해도 상관이 없습니다.

기본적으로 C 언어 에서는 두 가지 형태의 주석을 지원합니다.

```
/*
 이렇게
 여러
 줄을
 걸쳐서
 쓸 수
 있는 주석과
*/

// 한 줄에만 쓸 수 있는 주석을 말이죠.
```

마치 컴파일러가 철저히 무시한다는 것을 반영하기라도 한 것인지, 주석은 초록색으로 표시됩니다. 보통 한 줄에 쓸 수 있는 주석은 // 로 나타내고, 주석이 조금 길어져 여러 줄에 걸쳐 표시하려면 /*

2) 근데 사실 그렇게 크게 신경쓰지 않아도 됩니다. 적어도 윈도우에서는 해당 리턴값은 그냥 무시됩니다.

와 */ 를 이용합니다. 한 번, 위 Hello, World! 프로그램에 자기 나름대로 주석을 넣어 설명을 해보세요.

뭘 배웠지?

- #include <stdio.h> 란 stdio.h 라는 파일을 포함하라는 뜻입니다. 이 파일에는 여러분이 화면에 메세지를 띄울 수 있게 도와주는 여러가지 함수들이 포함되어 있습니다.
- main 함수는 프로그램이 시작되는 **함수** 입니다.
- {} 는 함수의 몸체를 알려주는데 사용됩니다.
- printf 는 화면에 내용을 출력해주는 **함수** 입니다.
- return 0; 는 0 을 반환한다는 의미 입니다.
- // 와 /* */ 는 주석으로 컴파일러가 무시하지만, 프로그래머를 위해 남겨놓는 코멘트 입니다.

주석(Comment)

사실, 2 - 1강에서도 다룬 내용이지만 댓글을 통해 질문이 들어 왔기에 정확히 주석이란 놈이 무엇인지 알아 보도록 하겠습니다.

우리가 프로그래밍을 하다 보면 소스 코드가 상당히 길어 지게 됩니다. 우리가 앞서 한 `Hello, World!` 출력 예제는 소스 코드가 겨우 몇 줄에 불과하였지만 실제론 소스 코드의 길이가 수천 줄에서 수만 줄 가까히 됩니다. 예를 들어서 우리가 지금 사용하는 `Windows XP` 의 소스 코드가 몇 줄 정도 될 지 추측해 보세요. 한, 십만줄? 50만 줄? 아닙니다. 정확한 자료는 아니지만 대략 4000만 줄 이상 된다고 합니다.

이런 크기로 프로그램을 작성하다 보면 이 소스 코드가 무엇을 뜻하고 또 무슨 일을 하는지 등의 정보를 소스 코드 내에 나타내야 할 필요성이 있게 됩니다. 즉, 컴파일러가 완전히 무시하고 오직 사람의 편의를 위해서만 존재하는 것이 바로 **주석** 입니다.

종종 지금 코드를 쓰고 있는 시점에서 내용을 잘 안다고 주석을 생략하는 경우가 있습니다. 하지만 주석 없는 코드를 1 달 뒤에 다시 본다면 분명히 아니 이게 지금 뭐하는 코드지? 라고 생각하실 것입니다. 반면에 주석이 잘 작성되어 있는 코드는 몇 년 뒤에서 다시 읽는다 해도 (주석을 잘 달아놨다는 가정 하에) 쉽게 이해할 수 있습니다.

C 언어에서 주석은 두 가지 방법으로 넣을 수 있습니다.

```
/* 주석이 들어가는 부분 */

// 주석이 들어가는 부분
```

일단 전자의 경우 `/*` 와 `*/` 로 묶인 내부의 모든 내용들이 주석으로 처리 됩니다. 즉,

```
/*

이 부분은 내가 아무리 생쇼를 해도 컴파일러가 무시
ㅁㄴㅇ러ㅏㅁ니ㅓㄹ먼리;ㅏㄴㅁ어림나러 무시
ㅁ아ㅓㄹ민럼ㄴㄹ ㅁㄴㅇ럼니러
ㅁㄴ라미너럼나러ㅣㅁ너라ㅣㅓㅇ
printf("Hello, World"); <- 이 것도 당연히 무시
*/
```

와 같이 난리를 쳐도 `/*` 와 `*/` 로 묶인 부분은 무시됩니다. 아래 예제를 보면 이해가 더욱 잘 될 것입니다.

```
#include <stdio.h>
int main() {
  /*
```

```
  printf("Hello, World!\n");
  printf("Hi, Human \n");
  */
  printf("Hi, Computer \n");

  return 0;
}
```

위와 같은 소스코드를 컴파일 하였을 때,

실행 결과

```
Hi, Computer
```

와 같이 `Hi, Computer` 을 출력하는 부분만 남을 것을 볼 수 있습니다. 이는 앞서 말했듯이 `/*` 와 `*/` 로 묶인 부분이 전부다 주석으로 처리되어서 컴파일러가 철저하게 무시하였기 때문 입니다.

반면의 `//` 형태의 주석의 경우 `//` 가 쳐진 줄 만이 주석으로 처리가 됩니다. 즉,

```
// Hello, World! 를 출력한다.
printf("Hello, World!");
```

로 하면 아래 `printf` 부분 잘 실행됩니다. 하지만 위 주석은 역시 무시됩니다. 아래 예제를 보면 확실히 알 수 있습니다.

```
#include <stdio.h>
int main() {
  // printf("Hello, World!\n");
  printf("Hi, Human \n");
  printf("Hi, Computer \n");

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Hi, Human
Hi, Computer
```

와 같이 주석으로 감싸진 부분을 제외하고는 나머지 부분이 잘 출력되었음을 알 수 있습니다.

뭘 배웠지?

- 항상 주석을 다는 습관을 기릅시다.
- 주석을 잘 달지 않는다면 나중에 자기가 쓴 코드도 못알아보게 됩니다.

컴퓨터에서 수를 표현하는 방법

아마 제 강좌를 읽는 분들 중에선 중학교 수학을 접하지 않는 분들도 있을 수 있기 때문에 이 강좌를 작성하게 되었습니다. 3 강에서 이 부분에 대한 내용을 다루고 있으나 질문이 자주 나오는 것 같아서 이렇게 강좌를 올립니다. 사실 이 부분의 내용은 중학교 수학에서는 모두 다루는 내용이므로 꼭 보셔도 되지 않지만 복습 차원에서 한 번쯤 보아도 좋습니다. 또한, 앞으로 중요하게 쓰일 몇 가지 컴퓨터 용어들을 다루고 있으니 (물론 3 강에서도 나옵니다만...) 한 번 보는 것이 좋겠습니다.

여러분은 '수' 와 '숫자' 의 차이를 알고 계십니까? 아마 많은 사람이 모를 텐데 영어로 하면 '수' 는 Number 이고 '숫자' 는 Digit 라 합니다.

'수' 는 어떠한 물질의 양을 나타내는 단위 입니다. '숫자' 는 이를 기록할 수 있도록 시각화 한 것이지요. 예를 들어서, 사과가 100 개 있다는 사실을 보여주기 위해 사과를 100 개 그려야 되면 상당히 곤란하겠지요. 단순히 '100' 이라는 것만 써서 사과가 100 개 있다는 사실을 알 수 있습니다.

그런데 놀라운 점은 우리가 이러한 `0,1,2,3,4,5,6,7,8,9` 의 숫자들만을 이용하여 수를 표시할 수 있다는 것이 아닙니다. 우리가 사과 라는 물질을 나타내기 위해 '사과' 라고 쓰지만 사과를 나타내기 위해 꼭 '사과' 라고 써야 하나요? 영어로 'Apple' 이라 쓸 수 도 있고 일본어로 'リンゴ' 라고 쓸 수 있습니다. 수도 마찬가지 입니다. 인간의 손가락이 10개 인지라 수를 10 개의 숫자로 나타내었지만 마음에 안들면 0 과 1 만을 이용할 수 도 있고 100 개의 숫자를 이용할 수 도 있습니다.

이렇게 수를 표현하는 방법을 **기수법(Numeral system)** 이라 합니다.

수학적 배경 지식 - 밑과 지수

여기서 소개하는 내용은 중학교 1 학년 재학시 배우게 됩니다만, 제 블로그를 방문하는 분들 중에 초등학생들도 적지 않게 있으리라 생각되어 간단히 짚고 넘어가겠습니다.

수학은 '복잡한 것을 단순히!' 라는 목적 하에 발전해 왔습니다. 이것도 이러한 본분을 충실히 따른 것 입니다. 우리는 곱하기 연산을 자주 사용합니다. 그 때 마다 이를 표현하기 위해 곱하기 기호를 사용해야 합니다. 예를 들어 아래 처럼 2 를 100 번 곱한 수를 생각해 봅시다.

$$2 \times 2 \times 2 \times \cdots \times 2$$

(너무 길어서 약간 생략)

우리는 위 수를 나타내고 싶을 때 마다 위와 같이 2 를 100 번이나 적어야 합니다. 다행이도 수학자들은 위를 단순화 하여 나타내기 위해 아래와 같은 기호를 만들어 냈습니다.

$$2^{100} = 2 \times 2 \times 2 \times \cdots \times 2$$

정말 훌륭한 생각 아닙니까? 단순히 2 를 100 번 곱했다는 사실을 알려주기 위해서 2 위에 작은 글씨로 100 을 적었습니다. 이 때 아래 '2' 를 '밑(base)' 라 부르고 밑 위에 밑을 몇 번 곱할 지 나타낸 수 100을 '지수(exponent)' 라고 부릅니다. 위 숫자의 경우 밑은 2 가 되고 지수는 100 이 되겠지요.

아마 이해가 잘 안되면 아래와 같은 예를 보시면 됩니다.

$$3^5 = 3 \times 3 \times 3 \times 3 \times 3$$

$$7^2 = 7 \times 7 = 49$$

지수에서 가장 중요한 사실은 바로

$$(\text{Number})^0 = 1$$

라는 사실입니다. `Number` 에는 어떠한 수도 들어 갈 수 있습니다. (다만 0 제외) 아마 이 부분에서 의아해 하는 사람들이 많을 것 입니다. "어떻게 숫자를 한 번도 안 곱했더니 1 이 될수가 있냐! 수학자 바보들 아니냐!" 말이지요. 하지만 사실 이 값은 사실 수학자들 사에서 약속 처럼 정의한 값이라 생각하면 됩니다. 만일 이 값을 1 이라 하지 않는다면 여러 지수들에 관련된 중요한 법칙들이 성립되지 않기 때문이죠. 다시말해 어떤 수의 0 승 한 것이 1 이라 생각하는 것은 다른 법칙들이 만족되기 위해 그렇게 약속한 것이다 정도로 알고 계시면 되겠습니다.

십진법, 이진법, 16 진법

현재 아라비아 숫자를 사용하는 우리는 수를 나타내기 위해 10 개의 숫자를 이용하는, 소위 말하는 **십진법(decimal)** 을 이용하고 있습니다. 정확하게 알려진 것은 아니지만 우리가 '10' 에 그토록 관련이 많은 것이 인간이 10 개의 손가락을 가졌기 때문이라 생각합니다. 아무튼, 우리는 '253' 이란 숫자가 나오면 아래와 같이 생각합니다.

$$253 = 2 \times 10^2 + 5 \times 10^1 + 3 \times 10^0$$

다시 쓰면,

$$253 = 200 + 50 + 3$$

이 됩니다. 한 가지 주목하실 부분은 바로 한 자리수가 늘어 날 때 마다 그 자리를 나타내는 숫자에 '10' 이 곱해진 다는 것이지요.가장 오른쪽 자리는 1 의 자리, 그 다음 자리는 10 의 자리, 그 다음은 100 의 자리, 그 다음은 1000 의 자리로 말이지요.예를 들어서 253 앞에 7 을 붙인다면

$$7253 = 7 \times 10^3 + 2 \times 10^2 + 5 \times 10^1 + 3 \times 10^0 = 7000 + 200 + 50 + 3$$

가 됩니다. 이 것이 바로 십진법의 가장 큰 특징이라고 할 수 있습니다. 자리수가 하나 증가할 때 마다 이 자리를 나타내는 숫자에 10 이 곱해지게 되지요. 또한 중요한 특징으론 십진법이 10 개의 숫자를 사용한다는 것 입니다. 0 부터 9 까지 모두 10 개의 숫자를 이용하지요.

이런 아이디어를 적용 시켜서 이진법을 생각해 봅시다. 컴퓨터는 0 과 1 인 두 종류의 숫자 밖에 표현할 수 없습니다. (전기적 신호가 *on* 이냐 *off* 냐에 따라 말이지요) 그러면 컴퓨터는 253 을 어떻게 생각할까요? 컴퓨터는 0 과 1 밖에 생각 못하므로 표현할 수 없다구요? 아닙니다. 컴퓨터도 0 과 1 만을 가지고서 모든 수를 표현 할 수 있습니다. 우리가 10 개의 숫자를 가지고 수를 표현하므로 이를 '십진법' 이라 부르지만 컴퓨터는 두 개의 숫자 만으로 수를 표현하므로 '이진법(Binary)' 라 부릅니다.

예를 들어 6 을 이진수로 나타낸다고 합시다. 그렇다면 우리가 십진법에서 사용한 규칙을 그대로 적용하여 사용하면 아래와 같습니다.

$$6 = 4 + 2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 110_2$$

어때요? 규칙이 정확히 동일한가요? 십진법에선 한 자리 늘어날 때 마다 10 이 곱해졌으므로 이진법에선 한 자리 늘어날 때 마다 2 가 곱해집니다. 즉, 가장 오른쪽 자리는 1, 그 다음은 1 에 2 를 곱했으므로 2, 그 다음은 2 에 2를 곱했으므로 4의 자리가 됩니다.

또한 십진법이 0 부터 9 까지의 수를 쓴 만큼 이진법도 0 과 1 만을 사용해야 합니다. 위에 보면 알 수 있듯이 6 을 110 으로 표현하였습니다. 즉, 1 과 0 만을 사용하였습니다.

110 밑에 조그맣게 2 가 표시되어 있는 것이 무엇인지 궁금해 하는 사람들이 있습니다. 이는 '이 수가 이진법으로 표현되어 있습니다' 를 알려주는 기호 입니다. 즉 2 라는 표시가 없다면 이 수가 정말 십진법으로 110을 표현한건지, 아니면 이진법으로 110을 표현한 것인지 알 길이 없기 때문이죠.

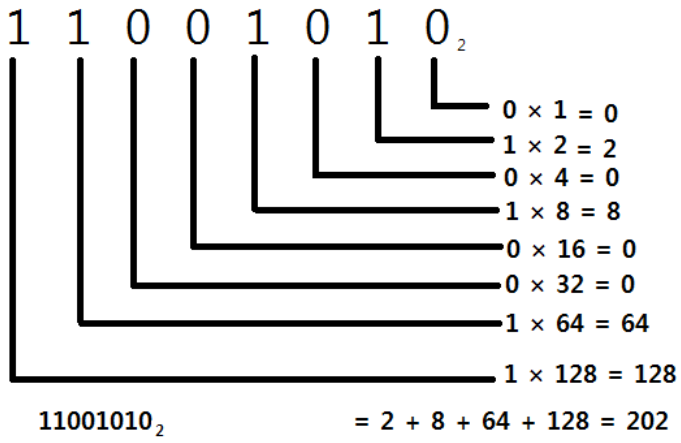
이진수가 무엇인지 확실히 감을 잡기 위해 아래의 수들을 보시면 됩니다.

$$23 = 16 + 4 + 2 + 1 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 10111_2$$

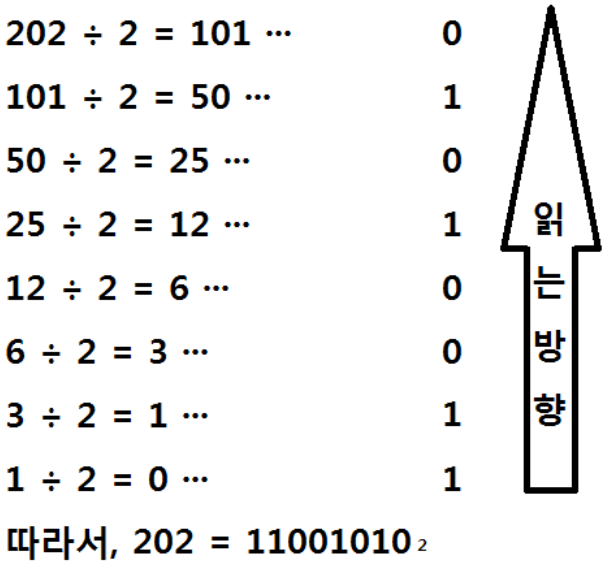
$$49 = 32 + 16 + 1 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 110001_2$$

그렇다면 이제 이진수를 어떻게 십진수로 바꿀 수 있는지 알아 봅시다. 사실, 위의 내용을 잘 이해하였다면 정말 간단합니다.

아래 그림과 같이 하면 됩니다.



자리수가 하나 올라갈 때 마다 그 자리수의 값이 두 배로 된다는 사실만을 기억하면 됩니다. 하지만 관건은 십진수를 이진수로 어떻게 쉽게 바꾸느냐 입니다. 사실 이는 어렵게 느껴지지만 그 과정은 매우 단순합니다. '2로 나누는 연산' 을 반복해 주면 되지요. 아래 그림을 참조하세요.



ůůů 의 기호는 나머지를 뜻합니다. 즉, 25 를 2 로 나누면 몫이 12 고 나머지가 1 이 됩니다. 단순히 십진수를 2 로 계산 나누어서 나온 나머지를 몫이 0 이 될 때 까지 구한 다음에 나머지들만 역순으로 재배치 하면 됩니다. 의외로 단순합니다.

(사실 위 방법이 어떻게 정확한 결과를 도출해 내느냐에 대해 설명해야 하지만 사실 세세히 알 필요도 없고 나중에 다 배우므로 생략하도록 하겠습니다.)

하지만, 이진수는 흔히 수가 너무 커지는 경향이 있습니다. 예를 들어 1024 는 십진수로 4 자리 밖에 안되지만 이진수로는 10000000000 이 되어 무려 11 자리나 됩니다. 1 과 0으로 나타내면 (사람들의

입장에서) 읽기 번거럽기 때문에, 이를 손쉽게 표현하기 위해서 프로그래머들은 보통 16진법을 사용하고 있습니다. 16 진법도 여태까지 사용하였던 아이디어를 동일하게 적용하면 됩니다.

그런데 한가지 주목해야하는 사실은 16 진수가 숫자를 16 개나 필요로 한다는 점입니다. 우리가 사용하는 십진법은 숫자가 10 개만 필요하므로 숫자가 10 종류 밖에 없습니다. 따라서 사람들은 16 진수를 위해 필요한 숫자 6개를 알파벳을 이용하여 숫자를 표현하였습니다. 즉 10 에 대응되는 것이 A, 11 에는 B, 12 에는 C, 13 에는 D, 14 에는 E, 15 에는 F 입니다. 16 진수는 **0,1,2,...,9,A,B,C,D,E,F** 를 이용하여 숫자를 표현 합니다.

$$123 = 7 \times 16 + 11 = 0\text{x}7\text{B}$$

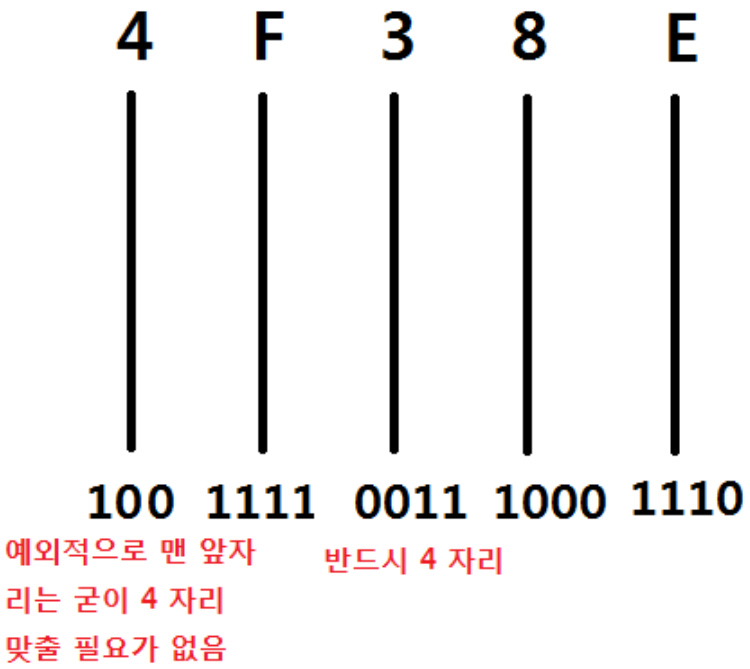
16 진수도 마찬가지로 같은 논리를 적용합니다. 한 자리가 늘어날 때 마다 16 을 곱하면 되는 것이지요. 이 때, `7B` 앞에 붙은 `0x` 는 이 수가 16 진수로 나타나있다는 것을 알려줍니다. 좀 더 이해를 돕기 위해 몇 가지 예를 들겠습니다.

$$19 = 16 + 3 = 0\text{x}13$$

$$16782 = 4 \times 16^3 + 1 \times 16^2 + 8 \times 16^1 + 14 \times 16^0 = 0\text{x}418\text{E}$$

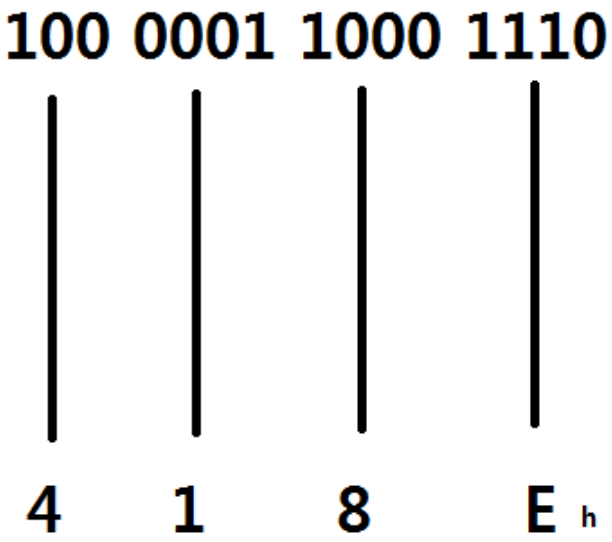
16 진수를 십진수로 바꾸거나 십진수를 16 진수로 바꾸는 일은 앞서 이진수의 경우와 동일합니다. (다만, 2 대신 16 으로 곱하거나 나누어 주어야 한다는 점 말고는) 이는 단순하지만 계산이 지저분하고 복잡하였죠. 하지만 16 진수를 이진수로, 이진수를 16 진수로 바꾸는 방법은 매우 간단합니다.

먼저 16 진수를 이진수로 바꾸는 방법을 살펴 봅시다. 이는 단순히 16 진수의 각 자리수를 4 자리 (반드시) 이진수로 변환해 주면 됩니다.



위와 같이 `0x4F38E` 를 2 진수로 변환하면 `10011110011100011110` 가 됩니다.

마찬가지로 이진수를 16 진수로 바꾸어 주는 것도 간단합니다. 단지 4 자리씩 뒤에서 부터 끊어서 읽으면 되지요. 아래 그림을 보면 알 수 있습니다.



음, 이제 어느정도 진법 체계에 통달하셨다고 생각합니다. 컴퓨터 계산기에 진법 변환 기능이 있으

므로 혼자서 연습해 보는 것도 좋을 것 같습니다. 그렇다면 이제 컴퓨터 메모리와 이진법과의 상관관계를 알아 보도록 합시다.

컴퓨터 메모리의 단위

컴퓨터에 대해 조금이나마 공부한 사람이라면 컴퓨터에 데이터를 저장하는 공간은 크게 두 부류로 나눌 수 있다고 들었을 것입니다. 컴퓨터를 종료하면 데이터가 날아가는 휘발성 메모리와 컴퓨터를 종료해도 데이터가 날아가지 않는 비휘발성 메모리로 말이지요.

이 때, 휘발성 메모리의 대표적인 주자로 램(RAM, Random Access Memory) 과 비휘발성 메모리의 대표 주자로 롬(ROM, Read Only Memory 흔히 말하는 CD - ROM 이나 DVD - ROM, 아니면 우리 메인보드 Bios 에 박혀있는 롬 등등) 나 하드 디스크 등이 있겠지요.

이 때, 앞으로 저의 강좌에서 주로 '메모리' 라 말하는 것은 휘발성 메모리인 RAM 을 말합니다. RAM 은 하드 디스크나 CD 와는 달리 속도가 매우 빠릅니다. (CD 휭 돌아가는 소리 들어 보셨죠?) 왜냐하면 RAM 의 경우 데이터의 랜덤하게 접근할 수 있는데 하드 디스크나 CD 는 순차적으로 접근해야 되기 때문이죠.

쉽게 말해 우리가 아파트 713 호에 사는 철수를 찾는다고 합시다. 철수가 만일 `RAM` 아파트에 산다면 단박에 713 호에 산다는 것을 알 수 있지만 하드 디스크 아파트에 산다면 101 호 부터 모든 주민들 일일히 찾아 713 호 까지 찾아 보아야 한다는 뜻 입니다.

이런 매우 빠른 메모리의 특성 때문에 컴퓨터는 대부분의 데이터들은 메모리에 보관해 놓고 작업을 하게 됩니다. 물론 메모리는 전원이 꺼지면 모두 날아가기 때문에 중요한 데이터들은 틈틈히 하드 디스크에 저장하게 되지요.

컴퓨터의 한 개의 메모리 소자는 0 혹은 1 의 값을 보관할 수 있습니다. 이 이진수 한 자리를 가리켜 비트(Bit) 라고 합니다. 따라서, 1 개의 비트는 0 또는 1 의 값을 보관할 수 있겠지요. 하지만 이는 너무나 작은 양입니다. 보통 우리는 1 보다 훨씬 큰 수들을 다루기 때문이지요. 그래서, 사람들은 이렇게 8 개의 비트를 묶어서 **바이트(Byte)** 라고 부릅니다. 즉, 8 비트는 1 바이트 이지요.

(참고로 4 비트를 특별히 묶어서 니블(nibble) 이라 부릅니다만, 잘 쓰이지는 않습니다)

8 비트로 나타낼 수 있는 수, 다시말해 8 자리 이진수로 나타낼 수 있는 최대의 수는 아래와 같이

$$00000000_2 \sim 11111111_2 = 0 \sim 255 = 0 \sim 0\text{xFF}$$

0 부터 255 로 총 256 개의 수를 나타내게 됩니다.

그 다음 단위로 **워드(Word)** 라고 부르는 단위가 있습니다. 컴퓨터에서 연산을 담당하는 CPU 에는 **레지스터(register)** 라는 작은 메모리 공간이 있는데, 이곳에다가 값을 불러다 놓고 연산을 수행하게 됩니다.

예를 들어서 `a + b` 를 하기 위해서는 `a` 와 `b` 의 값을 어디다 적어놓아야지, `a + b` 를 할 수 있는

것처럼, CPU 에서 연산을 수행하기 위해 잠시 써놓는 부분을 레지스터라고 합니다.

이러한 레지스터의 크기는 컴퓨터 상에서 연산이 실행되는 최소 단위라고 볼 수 있고, 이 크기를 **워드** 라고 부릅니다. 32 비트 컴퓨터 시절에서는 이 1 워드가 32 비트, 즉 4 바이트 였지만, 지금 대다수의 여러분이 사용하는 64 비트 컴퓨터의 경우 1 워드가 64 비트, 즉 8 바이트가 됩니다.[3)]

이것으로 여러분이 기수법과 컴퓨터 메모리의 단위에 대해 알아보았습니다. 그럼, 이번 강의는 여기서 마치도록 하죠.

뭘 배웠지?

- 이진법은 0 과 1 로, 십진법은 0 부터 9로, 16진법은 0 부터 9, A, B, C, D, E, F 로 수를 표현합니다.
- 1비트는 이진수로 숫자 1 개를 의미하며, 1 바이트는 8 비트, 즉 이진수로 8 자리 수를 의미합니다. 1 바이트로 0 부터 255 까지의 수를 표현할 수 있습니다.
- 컴퓨터에서 데이터를 잠시 기록해 놓는 것이 바로 메모리, 흔히 RAM 이라고 하는 곳입니다.
- 여러분이 사용하는 컴퓨터의 경우 대부분 4 바이트 혹은 8 바이트 단위로 데이터를 처리합니다.

3) 혹시 Windows API 를 공부한 분들이라면, Windows API 에 WORD 와 DWORD 로 정의된 키워드들을 생각하실 수 있습니다. 이 경우 WORD 는 16 비트, DWORD 는 32 비트를 나타냅니다.

2

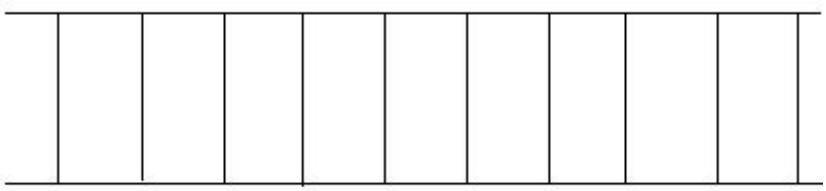
변수 (variable)

안녕하세요? 여러분. 잘 지내셨나요. 지난 번에 처음으로 C 코드를 분석한 것은 이해가 잘 되셨나요? 이해가 잘 안되셨다도 괜찮습니다. 점점 C 언어를 배워감에 따라, 기존이 이해가 안 되었던 것들도 언젠가는 '아 이래서 그랬구나' 하는 순간이 오게 됩니다. 이 단계에서 여러분이 취해야 할 자세는 일단 이해가 안되는 것은 일단, **암기** 하고, 포기하지 않는 것이 필요합니다.

변수란 무엇인가?

컴퓨터는 많은 내용을 기억 해야 합니다. 정확히 말하면, 컴퓨터의 '메모리' 라는 부분에 전기적인 신호를 써 놓는 것이죠. 컴퓨터가 무엇을 기억해야 되냐고 생각할 수 있지만, 우리가 많이 하는 게임인 스타크레프트만 보아도 일단, 각 유닛의 체력과 마나, 그리고 실시간으로 바뀌는 미네랄과 가스, 뿐만 아니라 유닛의 위치, 유닛의 데미지 등 모든 것을 기억해야지 우리가 게임을 제대로 즐길 수 있게 되겠지요. 만약 컴퓨터가 미네랄의 양을 제대로 기억 못한다면 미네랄이 갑자기 100 에서 0 이 되거나 10 에서 9999 로 바뀌는 참사가 발생합니다.

그렇다면 컴퓨터는 이러한 데이터들을 어떻게 기억할까요? 바로 컴퓨터의 메모리, 즉 **램(RAM) 이라는 특별한 기억공간**에 이를 기록합니다. 보통 우리는 흔히 램 을 설명할 때 아래 처럼 표시합니다.

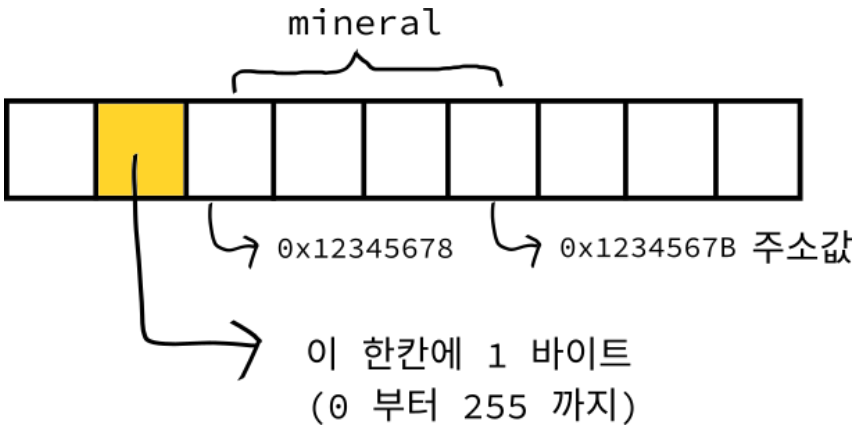


마치, 각 방에 데이터들이 저장됩니다. 이 때, 컴퓨터는 각 방에 이름을 붙이는데 단순하게 숫자로

이름을 붙입니다. 0 번, 1 번, 2 번, .. 와 같이 말입니다. 우리 대부분이 사용하는 32 비트 CPU 에서는 최대 2ş$ 개(4GB) - 총, 42 억개 달하는 방을 가질 수 있게 되겠지요. 참고로 32 비트 숫자를 매번 쓰는게 매우 힘들기 때문에, 대개 16진법으로 주소값을 나타냅니다.

예를 들어서, 컴퓨터가 `0x12345678` 부터 `0x1234567B` 부분에 내가 캔 미네랄의 양에 관한 정보를 저장했다고 합시다. (이 한칸에는 1 바이트, 즉 -128 부터 127 까지의 수 데이터를 저장할 수 있습니다) 만약 우리가 건물을 지을 때, 내가 가진 미네랄의 양이 충분한 지 확인하기 위해, 내가 캔 미네랄의 양에 관한 정보가 필요합니다. 그런데, 이렇게 미네랄에 관한 정보가 필요로 할 때 마다 이 길고 알아보기 힘든 복잡한 주소를 일일이 써야 한다면 상당히 힘들겠지요.[1)]

하지만 다행히도 C 언어에는 **변수** 라는 것이 있어서, 이 모든 작업을 쉽게 할 수 있습니다. 예를들어, 내가 캔 미네랄의 양을 `mineral` 이라는 변수에 저장했다고 합시다. 그렇다면 컴퓨터는 '알아서' 메모리의 어딘가에 `mineral` 의 방을 주고 그 내용을 저장합니다. 예를들어서, 컴퓨터가 이 `mineral` 이라는 변수에게 4 칸의 자리를 할당해 주었다고 합시다. 이는 아래 그림처럼 메모리 상에 표시됩니다.



이 때, 우리가 미네랄을 더 캐서 8 을 추가해야한다고 봅시다. 만약 이전에 8 을 추가한다면 `0x12345678` 부터 `0x1234567B` 까지의 모든 내용을 불러와서 8 을 더한 후, 다시 집어넣는 작업을 일일이 손으로 써 주어야 되었을 것입니다.

하지만, 이제는 단순히 `mineral = mineral + 8` 과 같이 써 주기만 한다면 `mineral` 에 8 이 더해지는 것이죠. (만약 `mineral = mineral + 8` 이라는 식이 이해가 안되도 그냥 넘어가세요. 이 처럼 간단해 진다는 것을 말해주고 싶었을 뿐입니다)

이와 같이 C 언어에서, **바뀔 수 있는 어떤 값을 보관하는 곳을 변수** 라고 합니다. 영어로는 *Variable* 이라 하는데, 말 그대로 바뀔 수 있는 것들 이라는 뜻입니다.

1) 0x12345678, 0x12345679, 0x1234567A, 0x1234567B 이렇게 4 개를 사용합니다.

변수 선언하기

```c
/* 변수 알아보기 */
#include <stdio.h>
int main() {
  int a;
  a = 10;
  printf("a 의 값은 : %d \n", a);
  return 0;
}
```

프로젝트를 만들어 위의 내용을 적은 후, 컴파일 해봅시다. 까먹었다면 1 강을 참조하세요. 만약 성공적으로 하였다면

실행 결과

```
a 의 값은 : 10
```

와 같이 나옵니다.

일단, 이번에도 역시 생소한 것들이 나왔기 때문에 한 문장씩 차근차근 살펴 봅시다.

```c
int a;
```

음, 이게 무엇일까요? 이전에 `int main()` 에서 보았던 `int` 가 다시 나타났군요.사실 이 문장에 뜻은 `a` 라는 변수를 우리가 쓰겠다고 컴파일러에게 알리는 것입니다. 만약 이러한 문장이 없다면 우리가 `x` 가 뭐고 `y` 가 뭔지 알려주지도 않은 채, 친구에게 `x + y` 가 얼마냐? 하고 물어보는 것과 똑같은 격이 되는 것이지요.

이 때, `a` 앞에 붙은 `int` 라는 것은 **int** 형의 데이터를 보관한다는 뜻으로, `a` 에 -2147483648 에서부터 2147483647 까지의 정수를 보관 할 수 있게 됩니다. 따라서, 만약 중간의 문장을

```c
a = 10000000000000;
```

와 같이 한다면 아마 `a` 의 값을 출력하였을 때, 이상한 결과가 나오게 됩니다. 왜냐하면 보관할 수 있는 범위를 초과하는 수를 보관했기 때문이죠.

그럼 이제, 걱정이 생깁니다. `a` 에 고작 10 밖에 안 넣을 거 면서, 굳이 2147483647 까지 표현할 수 있는 `int` 형의 변수를 왜 사용했냐고 물을 수 있습니다. 물론, `int` 형 보다 작은 범위의 숫자 데이터 만을 가지는 형식이 있기는 하지만 (`char` 등등), **일반적인 경우 정수 데이터를 보관할 때 `int` 형 변수를 사용합니다.**

또한, 2147483647 보다 큰 수를 사용하려면 어떻게 해야되냐는 궁금증도 생기지요. 물론 이 보다도 훨씬 큰 숫자를 처리하는 데이터 형식이 있습니다. 아래의 표를 참조하세요.

Name	Size*	Range*
char	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	1byte	true or false
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

http://www.cplusplus.com/doc/tutorial/variables/에서 가져왔습니다.

세번째 열인 Range 를 보시면, unsigned 와 signed 라고 나뉜 것이 있는데, 보통 int 라 하면 signed int 를 뜻합니다. 이는 음수와 양수 모두 표시할 수 있는 대신에 양수로 표현할 수 있는 범위가 줄어듭니다.

반면에 unsigned int 는 양수만을 표현할 수 있는 대신에, 양수로 표현할 수 있는 범위가 두 배로 늘어납니다. 또한 마지막에 보면 float, double, long double 이 있는데 이들은 '실수형' 자료형으로 소수(0.1, 1.4123 등) 을 표현 할 수 있습니다. 뿐만 아니라 double 의 경우, $\pm 2.3 \times 10^{-308} \backsim \pm 1.7 \times 10^{308}$ 의 수들을 표현 할 수 있습니다. (이에 대한 정확한 설명은 후에 다루겠습니다.)

```
a = 10;
```

위 문장은 무엇을 의미할까요? 언뜻 보기에도 감이 오시겠지만, 변수 a 에 10 을 집어넣는 다는 것입니다. 따라서 나중에 a 의 값을 출력한다면 10 이 나올 것입니다. 이와 같은 형태의 문장은 뒤에서 연산자에 대해 다룰 때 다시 알아보도록 하겠습니다.

```
printf("a 의 값은 : %d \n", a);
```

마지막으로, 지난번에도 보았던 printf 입니다. 그런데, 약간 다른 것이 있습니다. %d 가 출력되는 부분에 써져 있습니다. 그런데, 프로그램을 실행시켜 보았을 때 %d 는 컴퓨터에서 출력되지 않았습니다.

그 대신, %d 가 출력될 자리에 무언가 다른 것이 출력되었는데, 바로 a 의 값 입니다. 따라서,%d 는 **a 의 값 (정확히는 처음 "" 다음에 오는 첫 번째 변수) 을 *10* 진수 로 출력하라** 라는 뜻이 됩니다.

또 다른 예제를 봅시다.

```c
/* 변수 알아보기 2*/
#include <stdio.h>
int main() {
  int a;
  a = 127;
  printf("a 의 값은 %d 진수로 %o 입니다. \n", 8, a);
  printf("a 의 값은 %d 진수로 %d 입니다. \n", 10, a);
  printf("a 의 값은 %d 진수로 %x 입니다. \n", 16, a);
  return 0;
}
```

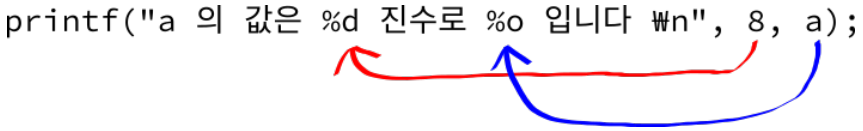
프로그램을 제대로 짰다면 아래와 같은 결과를 볼 수 있을 것입니다.

실행 결과

```
a 의 값은 8 진수로 177 입니다.
a 의 값은 10 진수로 127 입니다.
a 의 값은 16 진수로 7f 입니다.
```

일단, 위 코드를 보고 생기는 궁금증은 2 가지 있습니다. % 달린게 2 개나 있는데, 이를 어떻게 해야되냐와, %d 말고도 %o 와 %x 는 무엇인가 입니다.

먼저, printf 의 작동 원리에 대해 봅시다.



printf 출력시에, 큰 따옴표로 묶인 부분 뒤에 나열된 인자들 (8, a) 가 순서대로 큰 따옴표 안의 % 부분으로 들어감을 알 수 있습니다. 따라서 , 예를들면 printf("%d %d %d %d", a,b,c,d); 와 같은 문장은 a, b, c, d 의 값이 순서대로 출력되겠죠.

이제, %o 와 %x 는 무엇일까요? 이는 인자(a)의 값을 출력하는 형식 입니다. 즉, %o 는 a 의 값을 8 진수로 출력하라라는 뜻이고, %x 는 16 진수로 출력하라는 뜻 이죠.

실수형 변수

앞서 말했듯이, 실수형에는 float 와 double 이 있습니다. double 의 경우 int 형에 비해 덩치가 2 배나 크지만 그 만큼 엄청난 크기의 숫자를 다룰 수 있습니다. 그 대신, 처음 15 개의 숫자들만 정확하고 나머지는 10 의 지수 형태로 표현됩니다. 또한 float 과 double 의 장점은 소수를 표시할 수 있다는 점인데, 정수형 변수에서 소수를 넣는다면 (예를들어 int a; a = 1.234;), 소수 부분은 다 잘린 채, 나중에 a 의 값을 표시해 보면 1 이 나올 것 입니다.

```c
/* 변수 알아보기 3*/
#include <stdio.h>
int main() {
  float a = 3.141592f;
  double b = 3.141592;
  printf("a : %f \n", a);
  printf("b : %f \n", b);
  return 0;
}
```

실행해 본다면 아래와 같이 나오게 됩니다.

실행 결과

```
a : 3.141592
b : 3.141592
```

일단, 위 코드를 보면서 궁금한 점이 생기지 않았나요?

```c
float a = 3.141592f;
double b = 3.141592;
```

왜, `float` 형 변수 `a` 를 선언할 때 에는 숫자 뒤에 `f` 를 붙였는데 `double` 형 에서는 `f` 를 안 붙였는 지요. 왜냐하면, 그냥 `f` 를 안 붙이고 `float a = 3.141592` 로 하면 이를 `double` 형으로 인식하여 문제가 생길 수 있습니다. 따라서, `float` 형이라는 것을 확실히 표시해 주기 위해 `f` 를 끝에 붙이는 것입니다.

```c
printf("a : %f \n", a);
printf("b : %f \n", b);
```

이제, 마지막으로 `%d, %o, %x` 도 아닌 `%f` 가 등장하였습니다. 만약, 여기서 `a` 를 `%d` 형식으로 출력하면 어떻게 될까요? 한 번 해보세요. 아마 이상한 숫자가 나오게 될 것입니다. 왜냐하면 `a` 는 지금 정수형 변수가 아니기 때문 입니다. 설사, 우리가 `a = 3f; b = 3;` 라고 해도, 이미 `a` 와 `b` 를 실수형 변수로 선언하였기 때문에 컴퓨터는`a ,b` 를 절대 정수로 보지 않습니다.

따라서, 우리는 실수형 변수를 출력하는 형식인 `%f` 를 사용해야 합니다.

참고로 주의할 사항은 `printf` 에서 `%f` 를 이용해 수를 출력 할 때 다음과 같이 언제나 소수점을 뒤에 붙여 주어야 한다는 점입니다. 예를 들어서

```c
printf("%f", 1);
```

을 하면 화면에 이상한 값 (아마도 0 이 출력될 것입니다) 이 나오지만

```
printf("%f", 1.0);
```

을 하면 화면에 제대로 1.0 이 출력됩니다.

printf 의 또 다른 형식

```
/* printf 형식 */
#include <stdio.h>
int main() {
  float a = 3.141592f;
  double b = 3.141592;
  int c = 123;
  printf("a : %.2f \n", a);
  printf("c : %5d \n", c);
  printf("b : %6.3f \n", b);
  return 0;
}
```

만약 위 소스를 성공적으로 쳤다면 실행시 아래와 같이 나오게 됩니다.

실행 결과

```
a : 3.14
c :   123
b :  3.142
```

```
printf("a : %.2f \n", a);
```

이번에는 %f 가 아니라 %.2f 로 약간 다릅니다. 그렇다면 .2 가 뜻 하는 것은 무엇일까요? 대충 짐작했듯이, 무조건 소수점 이하 둘째 자리 까지만 표시하라 란 뜻입니다. 따라서, 위의 경우 3.141592 중 3.14 까지만 출력되고 나머지는 잘리게 되죠.

여기서 '무조건' 이라는 것은 %.100f 로 할 경우에도, 3.141592000000....00 을 표시해서 무조건 100 개를 출력하게 합니다.

```
printf("c : %5d \n", c);
```

이번에는 %d 가 아닌 %5d 입니다. 여기서 .5 가 아님을 주의합시다. 이 말은, 숫자의 자리수를 되도록 5 자리로 맞추라는 것입니다. 따라서, 123 을 표시할 때, 5 자리를 맞추어야 하므로 앞에 공백을 남기고 그 뒤에 123 을 표시했습니다.

그런데, 123456 을 표시할 때, %5d 조건을 준다면 어떻할까요? 이 때는 그냥 123456 을 다 표시합니다. 앞서 .?f 는 ? 의 수 만큼 무조건 소수점 자리수를 맞추어야 하지만 이 경우는 반드시 지켜야 되는 것은 아닙니다

```
printf("b : %6.3f \n", b);
```

마지막으로, 위에서 썼던 두 가지 형식을 모두 한꺼번에 적용한 모습입니다. 전체 자리수는 6 자리로 맞추되 반드시 소수점 이하 3 째 자리 까지만 표시한다는 뜻입니다.

변수 작명하기

앞서, 보았듯이 변수를 선언하는 것은 어려운 일이 아닙니다. 단지, 아래의 형태로 맞추어 주기만 하면 됩니다.

```
(변수의 자료형) 변수1, 변수2, .....;
/* 예를 들어 */
int a, b, c, hi;
float d, e, f, bravo;
double g, programming;
long h;
short i;
char j, k, hello, mineral;
```

이 때, 변수 선언시 주의해야 할 점이 있습니다. 만약에 여러분이 오래된 버전의 C 언어 (C89) 를 사용한다면, 변수 선언시 반드시 최상단에 위치해야 합니다. 하지만, 여러분이 지금 사용하고 있는 최신 버전의 C 의 변수 사용하기 전 아무데나 변수를 선언해도 상관 없습니다.

```
/* 변수 선언시 주의해야 할 점 */
#include <stdio.h>
int main() {
  int a;
  a = 1;
  printf("a 는 : %d", a);
  int b;  // 괜찮음!
  return 0;
}
```

두 번째로, 사람의 이름을 지을 때, 여러가지를 고려하듯이 변수의 이름에서도 여러가지 조건들이 있습니다. 아래 예제를 보세요.

```c
/* 변수 선언시 주의해야 할 점 */
#include <stdio.h>
int main() {
  int a, A;  // a 와 A 는 각기 다른 변수 입니다.
  int 1hi;
  // (오류) 숫자가 앞에 위치할 수 없습니다.
  int hi123, h123i, h1234324;  // 숫자가 뒤에 위치하면 괜찮습니다.
  int 한글이좋아;
  /*
  (오류)
  변수는 오직 알파벳, 숫자, 그리고 _ (underscore)로만으로 이루어져야 합니다. */
  int space bar;
  /*
  (오류)
  변수의 이름에는 띄어쓰기하면 안됩니다.  그 대신 _ 로 대체하는 것이 읽기
  좋습니다.*/
  int space_bar;  // 이것은 괜찮습니다.
  int enum, long, double, int;
  /* (오류)
     지금 나열한 이름들은 모두 '예약어' 로 C 언어에서 이미 쓰이고 있는
     것들입니다. 따라서 이러한 것들은 쓰면 안됩니다. 이를 구분하는 방법은
     예약어들을 모두 외우거나 '파란색' 으로 표시된 것들은 모두 예약어라 볼 수
     있습니다  */

  return 0;
}
```

이 안에 모든 내용이 들어 있습니다. 변수의 이름은 반드시

- 숫자가 앞에 위치하면 안됩니다. 그러나 중간이나 뒤는 괜찮습니다.
- 변수명은 오직 영어, 숫자, _ 로 만 구성되어 있어야 합니다.[2)]
- 변수의 이름에 띄어쓰기가 있으면 안됩니다.
- 변수의 이름이 C 언어 예약어 이면 안됩니다. 보통 예약어를 쓰면 에디터에서 다른 색깔로 표시되어 예약어를 썼는지 안썼는지 알 수 있습니다.

또한 C 언어는 대소문자를 구분합니다(이를 영어로 *case sensitive* 하다고 합니다). 따라서, `VARiable` 과 `Variable` 은 다른 변수 입니다. 왠지, 조건이 많아 변수명을 지을 때, 까다로울 것 같지만 그냥 평범하게 짓다보면 예약어와 겹칠일 도 없고, 숫자가 앞에 오는 경우도 별로 없습니다.

2) 사실 비주얼 스튜디오와 같은 컴파일러에서는 유니코드(한글 포함)로 변수 이름을 지어도 괜찮지만 관습상 코드는 모두 영어로 작성하는 것이 맞습니다.

자, 이제 우리는 C 언어에서 중요한 부분인 변수에 대해서 알아보았습니다. 현재 우리는 수를 다루는 변수들만 다루었지만, 다음 강좌에서는 변수에 대한 산술 연산과, 문자를 다루는 변수에 대해 알아보도록 하겠습니다.

뭘 배웠지?

- 변수는 데이터를 임시로 저장하는 곳이며 자유롭게 쓰고 지울 수 있습니다.
- 각 변수에는 형(type) 이 있어서 해당 형에 맞는 데이터를 보관할 수 있습니다.
- 변수의 형으로는 정수값을 보관하는 `char`, `int` 등이 있고, 실수값을 보관하는 `float` 과 `double` 이 있습니다. 각각의 형들은 저장하는 데이터의 크기가 다릅니다.
- `int a = 10;` 의 문장의 의미는 `a` 라는 정수형 변수를 정의한 뒤에, 해당 변수에 10 의 값을 대입한다 라는 뜻입니다.
- 변수의 이름을 정하기 위해서는 여러가지 규칙이 있습니다. 이 규칙에 알맞게 변수의 이름을 정해야 되며 그렇지 않을 경우 컴파일 오류가 발생합니다.

3

계산 하기

안녕하세요 여러분. 지난 강의에서 모두들 변수에 대해 감이 잡혔을 것이라 믿고 강의를 진행하도록 하겠습니다.

최초의 컴퓨터는 무엇을 하기 위해 태어났을까요? 오락용? 영화 시청? (물론 그 때에는 불가능했을 터이지만). 아닙니다. 최초의 컴퓨터라 일컫어 지는 에니악(ENIAC.. 물론 에니악이 최초의 컴퓨터이냐 아니냐에 관한 논쟁은 길다. 한편에서는 콜로서스라는 주장도 있는데 아무튼) 은 포탄을 어떤 각도로 발사했을 때, 어디에 떨어질 지를 예측하는 기계였습니다.

물론 지금도 컴퓨터의 가장 중요한 역할은 인간이 할 수 없는 복잡한 수식을 계산하는 것입니다. 다시말해, 컴퓨터는 **빠른 계산**을 위해 태어난 기계인 것입니다.

산술 연산자, 대입 연산자

이번 강좌에서는 C 언어에서 컴퓨터에 어떻게 연산 명령을 내리는지 살펴보도록 하겠습니다.

일단, '계산' 이라 하면 머리속에 가장 먼저 떠오르는 것은 사칙연산, 즉 +, −, Œ, œ 을 의미합니다. 보통 코딩 시에 Œ 와 œ 기호를 쓰기 힘들기 때문에, 그 대신 `*` 와 `/` 를 사용합니다. 즉, 8 Œ 5 는 `8 * 5` 로 표현하고, 10 œ 7 은 `10 / 7` 로 표현합니다.

또한, 색다른 연산자로 `%` 가 있는데 이는 나눈 나머지를 의미합니다. 예를들어 `10 % 3` 은 1 이 됩니다. 왜냐하면 10 을 3 으로 나눈 나머지가 1 이기 때문이죠. 이러한 `+, -, *, /, %` 를 **산술 연산자(Arithmetic Operator)** 라고 합니다.

```c
/* 산술 연산 */
#include <stdio.h>
int main() {
  int a, b;
  a = 10;
  b = 3;
  printf("a + b 는 : %d \n", a + b);
  printf("a - b 는 : %d \n", a - b);
```

```
  printf("a * b 는 : %d \n", a * b);
  printf("a / b 는 : %d \n", a / b);
  printf("a %% b 는 : %d \n", a % b);
  return 0;
}
```

만약 위 코드를 잘 컴파일 했다면 아래와 같이 나옵니다. (컴파일 하는 방법을 까먹은 사람들은 1강을 참조하세요)

실행 결과

```
a + b 는 : 13
a - b 는 : 7
a * b 는 : 30
a / b 는 : 3
a % b 는 : 1
```

그렇다면 코드를 살펴보도록 합시다.

```
a = 10;
b = 3;
```

지난 강좌를 잘 이해하셧더라면 위 문장이 무슨 역할을 하는지 쉽게 이해하실 수 있을 것입니다. a 라는 변수에 10 을 대입하는 것이고, 두 번째 문장은 b 에 3을 대입하는 것입니다. 그렇다면 아래 문장을 살펴보세요.

```
10 = a;
3 = b;
```

언뜻 보기에 맞는 문장인 것 같습니다. 왜냐하면, 실제 수학을 공부한 사람이라면 `a = 10` 이나 `10 = a` 나 별반 다를 것이 없기 때문이죠. 하지만, C 언어 컴파일러는 '=' 라는 기호를 뒤에서 부터 해석합니다. 즉, `a = 10` 은 *10* 을 *a* 에 대입하라 라는 문장이 되지만, `10 = a` 는' a 의 값을 10 에 대입하라' 라는 이상한 문장이 되서 오류가 발생하게 됩니다.

이렇게 '=' 를 **대입 연산자(Assignment Operator)** 라고 합니다. 왜냐하면 우측의 값을 좌측에 **대입** 하는 것이기 때문이죠.

따라서,

```
a = 5;
b = 5;
```

```
c = 5;
d = 5;
```

라는 문장이나,

```
a = b = c = d = 5;
```

라는 문장은 완전히 같은 것이 됩니다. 왜냐하면, 앞에서 말했듯이 = 는 뒤에서 부터 해석한다고 했으므로, 제일 먼저 d = 5 를 해석한 후, 그 다음에 c = d, b = c, a = b 로 차례대로 해석해 나가기 때문에 a = 5; b = 5; c = 5; d = 5; 라는 문장과 같은 것이지요.

```
printf("a + b 는 : %d \n", a + b);
printf("a - b 는 : %d \n", a - b);
printf("a * b 는 : %d \n", a* b);
printf("a / b 는 : %d \n", a / b);
printf("a %% b 는 : %d \n", a % b);
```

자, 이제 산술 연산자들에 대해 살펴보도록 합시다. 일단, 한 눈에 보게 a + b, a - b, a * b, a / b 는 각각 덧셈, 뺄셈, 곱셈, 나눗셈을 하여서 그 값이 %d 에 들어가 출력된 것 같습니다. 그런데, a + b, a - b, a * b 는 각각 계산 결과가 13, 7, 30 이 나온 사실을 쉽게 받아들일 수 있지만, a / b 가 왜 3 이 나왔는지는 이해하기 힘듭니다. 왜, a / b 가 3 이 되었을까요?

사실, 3 강에서 말했지만 a 와 b 는 모두 int 형으로 선언된 변수 입니다. 즉, a 와 b 는 오직 '정수' 데이터만 담당합니다. 즉, a 와 b 는 모두 정수 데이터만 처리하기 때문에 a 를 b 로 나누면, 즉 10 을 3 으로 나누면 3.3333... 이 되겠지만 정수 부분인 3 만 남기게 되는 것 입니다. 따라서, 값은 3 이 출력됩니다.

마지막으로 생소한 % 라는 연산자에 대해 살펴봅시다. +, -, *, / 연산자는 모두 정수, 실수형 데이터에 대해서 모두 연산이 가능한데, % 는 오직 정수형 데이터에서만 연산이 가능합니다. 왜냐하면 % 는 나눈 나머지를 표시하는 연산자 이기 때문이죠. a % b 는 a 를 b 로 나눈 나머지를 표시합니다. 즉, 10 % 3 = 1 이 됩니다.

이 때,

```
printf("a %% b 는 : %d \n \n", a % b);
```

%% 는 % 를 '표시'하기 위한 방법입니다. 왜냐하면 % 하나로는 %d , %f 같이 사용될 수 있기 때문이 표시가 되지 않습니다.

나눗셈 시에 주의할 점

```c
#include <stdio.h>
int main() {
  int a, b;
  a = 10;
  b = 3;
  printf("a / b 는 : %f \n", a / b);  // 해서는 안될 짓
  return 0;
}
```

컴파일 후 (아마 경고 메세지가 뜰 것입니다), 실행한다면 아래와 같이 이상한 결과가 나옵니다.

실행 결과

```
a / b 는 : 0.000000
```

3 강 에서 우리는 %f 가 오직 실수형 데이터 만을 출력하기 위해 있는 것이라 하였습니다. 그런데, a / b 가 10 나누기 3 이므로 3.3333... 이 되서 해서 실수형 데이터가 되는 것이 아닙니다.

(정수형 변수) (연산) (정수형 변수) 는 언제나 (정수) 으로 유지됩니다. 따라서, 실수형 데이터를 출력하는 %f 를 정수형 값 출력에 사용하면 위와 같이 이상한 결과가 나오게 됩니다.

그렇다면 아래의 경우 어떻까요?

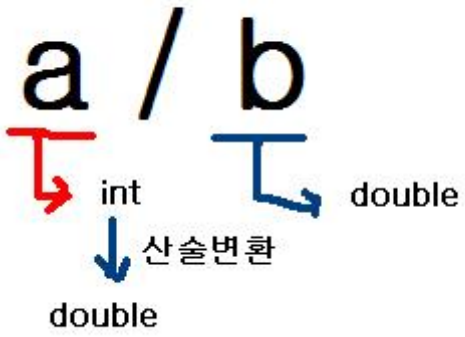
```c
/* 산술 변환  */
#include <stdio.h>
int main() {
  int a;
  double b;

  a = 10;
  b = 3;
  printf("a / b 는 : %f \n", a / b);
  printf("b / a 는 : %f \n", b / a);
  return 0;
}
```

만약 제대로 컴파일 했다면 아래와 같이 나오게 됩니다.

실행 결과

```
a / b 는 : 3.333333
b / a 는 : 0.300000
```

a 는 정수형 변수, b 는 실수형 변수 입니다. 그런데, 이들에 대해 연산을 한 후에 결과를 실수형으로 출력하였는데 정상적으로 나왔습니다. 그 것은 왜 일까요? 이는 컴파일러가 **산술 변환** 이라는 과정을 거치기 때문입니다. 즉, 어떠한 자료형이 다른 두 변수를 연산 할 때, 숫자의 범위가 큰 자료형으로 자료형들이 바뀝니다.



즉, 위 그림에서도 보듯이 a 가 int 형 변수이고 b 가 double 형 변수인데, double 이 int 에 비해 포함하는 숫자가 더 크므로 큰 쪽으로 산술 변환됩니다.

일단, 정수형 변수와 실수형 변수가 만나면 무조건 실수형 변수쪽으로 상승되는데, 이는 실수형 변수의 수 범위가 int 보다 훨씬 넓기 때문입니다. 위와 같은 산술 변환을 통해 애러가 없이 무사히 실행 될 수 있었습니다. 또한 double 형태로 산술 변환 되므로 결과도 double 형태로 나오기 때문에

```
printf(" a / b 는 : %d \n", a / b);
```

와 같이 하면 오류가 생기게 됩니다. 왜냐하면 %d 는 정수형 값을 출력하는 방식이기 때문이죠.

대입 연산자

```
/* 대입 연산자  */
#include <stdio.h>
int main() {
  int a = 3;
  a = a + 3;
  printf("a 의 값은 : %d \n", a);
  return 0;
}
```

위 결과를 컴파일 하면 아래와 같이 나옵니다.

실행 결과

```
a 의 값은 : 6
```

일단, 변수 선언 부분 부터 살펴 봅시다.

```
int a = 3;
```

위 문장을 아마 무슨 뜻일지 감이 바로 오실 것입니다. "음... a 라는 변수를 선언하고 a 변수에 3 의 값을 집어 넣는구나". 맞습니다. 사실 위 문장이나 아래 문장이나 다를 바가 없습니다.

```
int a;
a = 3;
```

그냥, 타이핑 하기 귀찮아서 짧게 써 놓은 것 뿐입니다.

```
a = a + 3;
```

그 다음 부분은 대입 연산자와 산술 연산자가 함께 나와 있습니다. 만일, 우리가 방정식에 대해서 공부해 본 사람이라면 다음과 같이 이의를 제기할 수 도 있습니다.

$a = a + 3$ 따라서 양변에서 a 를 빼면 $0 = 3$???

물론, 위는 수학적으로 맞지만 C 언어 에서 의미하는 바는 다릅니다. 위에서 말했듯이, = 는 등호가 아닙니다. '대입' 연산자 입니다. 무엇을 대입하냐구요? 오른쪽의 값을 왼쪽으로 대입합니다. 즉, a + 3 의 값(6) 을 a 에 대입합니다. 따라서, a = 6 이 되는 것이지요.

이 때, 이와 같이 계산 될 수 있는 이유는 + 를 = 보다 먼저 연산하기 때문입니다. 즉, a + 3 을 먼저 한 후(+), 그 값을 대입(=) 하는 순서를 거치기 때문에 a 에 6 이라는 값이 들어갈 수 있게 됩니다. 이러한 것을 **연산자 우선순위** 라고 하는데, 밑에서 조금 있다가 다루어 보도록 하겠습니다.

```
/* 더하기 1 을 하는 방법 */
#include <stdio.h>
int main() {
  int a = 1, b = 1, c = 1, d = 1;

  a = a + 1;
  printf("a : %d \n", a);
  b += 1;
  printf("b : %d \n", b);
  ++c;
```

```
  printf("c : %d \n", c);
  d++;
  printf("d : %d \n", d);

  return 0;
}
```

위 코드를 컴파일 하면 아래와 같이 나옵니다.

실행 결과

```
a : 2
b : 2
c : 2
d : 2
```

음, 모두 2 가 되었군요. 사실 위에 나온 4 개의 코드는 더하기 1 을 한다는 점에서 모두 같습니다. 일단, 하나하나 차례대로 살펴봅시다.

```
a = a + 1;
```

가장, 기초적으로 1 을 더하는 방법입니다. 위 문장은 "a 에 a 에 1 을 더한 값을 대입한다." 라는 뜻을 가지고 있죠?

```
b += 1;
```

이게 뭔가요! 처음 본 연산인 += 입니다. 이러한 연산을 복합 대입연산 이라 하며, b = b + 1 과 같습니다. 이렇게 쓰는 이유는 단지, b = b + 1 을 쓰기 귀찮아서 간략하게 쓰는 것입니다. 물론, b = b + 1 과 b += 1 은 엄밀히 말하자면 같은 것은 아니지만 이에 대해서는 나중에 다루어 보도록 하겠습니다(우선 순위에서 약간 차이가 있습니다). 복합 대입 연산은 아래와 같이 여러 가지 형태로 이용될 수 있습니다.

```
b += x; // b = b + x; 와 같다
b -= x; // b = b - x;와 같다
b *= x; // b = b * x;와 같다
b /= x; // b = b / x;와 같다
```

마지막으로, 비슷하게 생긴 두 부분을 함께 살펴 보도록 하겠습니다.

```
++c;
d++;
```

위와 같은 연산자(++)를 증감 연산자라고 합니다. 둘 다, c 와 d 를 1 씩 증가시켜 줍니다. 그런데, ++ 의 위치가 다릅니다. 전자의 경우 ++ 이 피연산자(c) 앞에 있지만 후자의 경우 ++ 이 피연산자(d) 뒤에 있습니다.

++ 이 앞에 있는 것을 **전위형(prefix)**, ++ 이 뒤에 있는 것을 **후위형(postfix)** 라 하는데 이 둘은 똑같이 1 을 더해주만 살짝 다릅니다. 전위형의 경우, 먼저 1 을 더해준 후 결과를 돌려주는데 반해, 후위형의 경우 결과를 돌려준 이후 1 을 더해줍니다. 이 말만 가지고는 이해가 잘 안될테니 아래를 보세요.

```c
/* prefix, postfix  */
#include <stdio.h>
int main() {
  int a = 1;

  printf("++a : %d \n", ++a);

  a = 1;
  printf("a++ : %d \n", a++);
  printf("a : %d \n", a);

  return 0;
}
```

위 소스를 성공적으로 컴파일 했다면 아래와 같이 결과가 나온다.

실행 결과

```
++a : 2
a++ : 1
a : 2
```

분명히, 위에서 ++c 나 d++ 이나 결과를 출력했을 때 에는 결과가 1 이 잘 더해져서 2 가 나왔는데 여기서는 왜 다르게 나올까요? 앞서 말했듯이 ++ a 는 먼저 1 을 더한 후 결과를 반환한다고 했고 a++ 은 먼저 결과를 반환 한 후, 그 후에 1 을 더한 다고 했습니다.

```c
printf("++a : %d \n", ++a);
```

즉, 위의 경우 a 에 먼저 1 을 더한 값인 2 를 printf 함수에 반환하여 %d 에 2 가 들어가게 됩니다. 그런데,

```c
printf("a++ : %d \n", a++);
```

이 경우, 먼저 a 의 값을 printf 에 반환하며 %d 에 1 이란 값이 '먼저' 들어 간 뒤, 1 이 출력된 이후 a 에 1 이 더해집니다. 따라서, 다시 printf 문으로 a 의 값을 출력하였을 때 에는 2 라는 값이 나오게 되는 것입니다. 참고로, 위 4 개의 연산 중에서 가장 빨리 연산되는 것은 a++ 과 같은 증감 연산입니다[1] 하지만, 요즘의 컴파일러는 최적화가 잘 되어 있어, a = a + 1 같은 것은 a++ 로 바꾸어 컴파일 해버립니다.

비트 연산자

마지막으로 **비트 연산자**라고 불리는 생소한 연산자들에 대해 이야기 해보겠습니다. 이 연산자들은 정말 비트(bit) 하나 하나에 대해 연산을 합니다. 비트는 컴퓨터에서 숫자의 최소 단위로 1 비트는 0 혹은 1 을 나타내죠. 쉽게 말해 이진법의 한 자리라 볼 수 있습니다.

보통, 8개의 비트(8 bit) 를 묶어서 1 바이트(byte) 라고 하고, 이진법으로 8 자리 수라 볼 수 있습니다. 따라서, 1 바이트로 나타낼 수 있는 수의 범위가 0 부터 11111111 로 십진수로 바꾸면 0 부터 255 까지 나타낼 수 있습니다.

비트 연산자에는 & (And 연산), | (\위에 있는 것. 영문자 i 의 대문자가 아닙니다. Or 연산), ^ (XOR 연산), <<, >> (쉬프트 연산) , ~ (반전) 등이 있습니다. 일단, 각 연산자가 어떠한 역할을 하는지 살펴보도록 합시다.

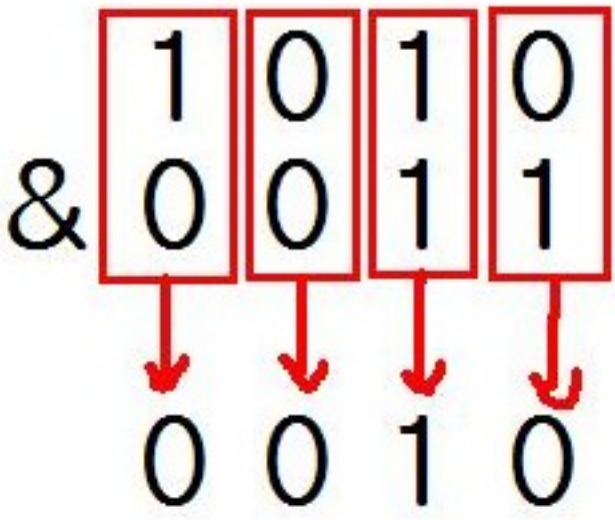
AND 연산 (&)

AND 연산은 아래와 같은 규칙으로 연산됩니다.

		결과
1	1	1
1	0	0
0	1	0
0	0	0

비트 연산은 각 자리를 연산하는데, 예를들어, 1010 & 0011 의 경우

1) 왜냐하면, a = a + 1 의 경우 ADD a 1 로 하지만, a++ 은 INC a 로 좀 더 빨리 계산된다. 자세한 내용은 나중에



위와 같이 한자리 한자리 각각 AND 연산하여, 위에 써 놓은 규칙대로 연산이 됩니다. 만약 두 숫자의 자리수가 맞지 않을 경우, 예를들어 1111100 과 11 을 AND 연산 할 때 에는 11 앞에 0 을 추가하여 자리수를 맞추어 줍니다. 즉, 1111100 과 0000011 의 연산과 같습니다.

OR 연산 (|)

		결과
1	1	1
1	0	1
0	1	1
0	0	0

OR 연산은 AND 연산과 대조적입니다. 어느 하나만 1 이여도 모두 1 이 되는데, 예를들어 `1101 | 1000` 은 결과가 1101 이 됩니다.

XOR 연산 ()

		결과
1	1	0
1	0	1
0	1	1
0	0	0

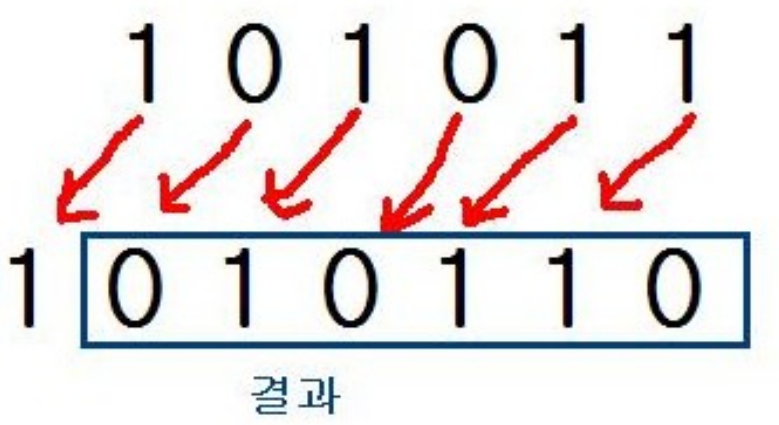
XOR 연산은 특이하게도 두 수가 달라야지만 1 이 됩니다. 예를들어, `1100 ^ 1010` 의 경우 결과가 0110 이 됩니다. 마치 두 비트를 더한 다는 식으로 생각하시면 됩니다.

반전 연산(~)

반전연산은 간단히 말에 0 을 1 로 1 을 0 으로 바꿔주는 것입니다. 예를들어서 `~1100` 을 하면 그 결과는 0011 이 됩니다.

« 연산 (쉬프트 연산)

위 연산 기호에서 느껴지듯이 비트를 왼쪽으로 쉬프트(Shift) 시킵니다. 예를 들어, 101011 를 1 만큼 쉬프트 시키면 (이를 `a << 1` 이라 나타냅니다)



위 처럼 결과가 010110 이 됩니다. 이 때, << 쉬프트 시, 만일 앞에 쉬프트된 숫자가 갈 자리가 없다면, 그 부분은 버려집니다. 또한 뒤에서 새로 채워지는 부분은 앞에서 버려진 숫자가 가는 것이 아니라 무조건 0 으로 채워집니다.

» 연산

이는 위와 같은 종류로 이는 << 와 달리 오른쪽으로 쉬프트 해줍니다. 이 때, 오른쪽으로 쉬프트 하되, 그 숫자가 갈 자리가 없다면 그 숫자는 버려집니다. 이 때, 무조건 0 이 채워지는 << 연산과는 달리 앞부분에 맨 왼쪽에 있었던 수가 채워지게 되죠. 예를들어서 `11100010 >> 3 = 11111100` 이 되고, `00011001 >> 3 = 00000011` 이 됩니다.[2)]

비트 연산자를 자세히 다룬 이유는 이 연산자가 여러분들이 아마 여태까지 살면서 다루어왔던 연산자들 (덧셈, 뺄셈 같은 애들) 과는 조금 다르기 때문입니다. 또한, 처음에 비트 연산자를 접할 때, 저런거 뭐에다 쓰지? 라는 생각이 들기도 합니다. 아직은 그 쓰임새를 짚고 넘어가기 어렵지만 나중에 종종 등장할 때가 있을 테니, 잘 기억해놓으시기 바랍니다.

```
/* 비트 연산 */
#include <stdio.h>
```

2) 참고로 어떤 시스템에서는 » 쉬프트의 경우에도 무조건 맨 왼쪽에 0 이 채워질 수 있습니다.

```c
int main() {
  int a = 0xAF;  // 10101111
  int b = 0xB5;  // 10110101

  printf("%x \n", a & b);   // a & b = 10100101
  printf("%x \n", a | b);   // a | b =10111111
  printf("%x \n", a ^ b);   // a ^ b = 00011010
  printf("%x \n", ~a);      // ~a = 1....1 01010000
  printf("%x \n", a << 2);  // a << 2 = 1010111100
  printf("%x \n", b >> 3);  // b >> 3 = 00010110

  return 0;
}
```

위를 성공적으로 컴파일 했다면

실행 결과

```
a5
bf
1a
ffffff50
2bc
16
```

위와 같이 나오게 됩니다. 일단, 첫 세줄은 그럭 저럭 이해가 잘 갑니다. 그런데, 네 번째 줄인 ~a 연산에 대해 의문을 품는 사람들이 많습니다.

```c
printf("%x \n", ~a);  // ~a = 1....1 01010000
```

우리의 기억을 되돌려 3 강으로 가 봅시다. 강의 중간쯤에 보면 여러가지 자료형 들에 대한 설명과 함께 작은 표가 있을 텐데 말이죠. 이를 다시 아래에 불러와 봅시다.

Name	Size*	Range*
char	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	1byte	true or false
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

`int` 형 변수에 대한 설명을 보니 옆에 **Size** 이라 표시된 것이 있습니다. 이는 `int` 형 변수의 크기를 나타내는데 4 바이트라고 되어 있군요. 맞습니다. `int` 형 변수는 하나의 데이터를 저장하기 위하여 메모리 상의 4 바이트 - 즉 32 비트를 사용합니다. (1 byte = 8 bits) 아까, 하나의 비트가 0 과 1 을 나타낸다고 했으므로 (즉 1 개의 비트가 2 진수의 한 자리를 나타내게 되죠), 하나의 `int` 형 변수는 32 자리의 이진수라고 볼 수 있습니다. 예를들어 우리가 `a = 1` 이라 한 것은 실제로 컴퓨터에는 `a = 00000000 00000000 00000000 00000001` 이라 저장되는 것과 같게 되는 거죠.

즉, 우리가 `int a = 0xAF;` 라고 한 것은 `a = 10101111;` 이 맞지만 사실 컴퓨터 메모리 상에서는 `a` 가 `int` 형이기 때문에 `a = 00000000 00000000 00000000 10101111` (`10101111` 앞에 0 이 24 개 있다) 이라 기억하는 것이 됩니다. 따라서, 이 숫자를 반전 시키게 되면`a = 11111111 11111111 11111111 01010000`, 즉 `0xFFFFFF50` 이 되는 것이지요. 마찬가지로 생각해 보면,

```
printf("%x \n", a << 2);  // a << 2 = 1010111100
printf("%x \n", b >> 3);  // b >> 3 = 00010110
```

이 두 문장도 사실은 각각 `00000000 00000000 00000000 10101111` 과 `00000000 00000000 00000000 10110101` 을 쉬프트 연산한 것과 같습니다.

따라서 `a` 의 경우 `00000000 00000000 00000000 10101111` 을 왼쪽으로 2 칸 쉬프트 하면 `00000000 00000000 00000010 10111100` 이 되어 `0x2BC` 가 됩니다

`b` 의 경우 마치 앞에 1 이 있는 것 같지만 실제로는 `int` 형 데이터에 저장되어 있으므로 4 바이트로, `00000000 00000000 00000000 10110101` 이므로 맨 왼쪽의 비트는 0 입니다. 따라서 쉬프트를 하게 되면 왼쪽에 0 이 채워지면서 `00000000 00000000 00000000 00010110` 이 되어 `0x16` 이 됩니다.

복잡한 연산

마지막으로 여러 연산이 중첩된 혼합 연산에 대해 살펴 보도록 합시다. 우리가 연산을 하는데 에도 순서가 있듯이 컴퓨터에도 연산을 하는데 무엇을 먼저 연산을 할 지 우선 순위가 정해져 있을 뿐더러 연산 방향 까지도 정해져 있습니다. 이를 간단히 살펴 보자면 아래와 같습니다.

1	( ) [ ] -> . (expr)++ (expr)--	왼쪽 우선
2	! ~ +-(부호) *p & sizeof 캐스트--(expr) ++(expr)	오른쪽 우선
3	*(곱셈) / %	왼쪽 우선
4	+ -(덧셈, 뺄셈)	왼쪽 우선
5	<< >>	왼쪽 우선
6	< <= > >=	왼쪽 우선
7	== !=	왼쪽 우선
8	&	왼쪽 우선
9	^	왼쪽 우선
10	|	왼쪽 우선
11	&&	왼쪽 우선
12	||	왼쪽 우선
13	? :	오른쪽 우선
14	= 복합대입	오른쪽 우선
15	,	왼쪽 우선

이와 같이 순위가 매겨져 있습니다. 이 때, 눈여겨 보아야 할 점은 괄호들이 제 1 우선 순위에 위치하였다는 점 입니다. 따라서, 어떠한 연산이라도 괄호로 감싸 주게 되면 먼저 실행 됩니다.

마지막으로, 결합 순위에 대해 잠시 다루어 보도록 하겠습니다. 표의 오른쪽을 보면 결합 순위가 나와 있는데, 대부분이 '왼쪽 우선' 이지만 몇 개는 '오른쪽 우선' 입니다. 이 말이 뜻하는 바가 무엇이냐면, 아래와 같은 문장을 수행할 때 계산하는 순위를 이야기 합니다.

```
a = b + c + d + e;
```

위 표에서 보듯이, 덧셈의 결합 순서가 왼쪽 우선이므로 위 계산과정은 아래의 순서대로 진행됩니다.

1. `b + c` 를 계산하고 그 결과를 반환(그 결과를 `C` 라 하면)
2. `C + d` 를 계산하고 그 결과를 반환(그 결과를 `D` 라 하면)
3. `D + e` 를 계산하고 그 결과를 반환(그 결과를 `E` 라 하면)

따라서, 위 식은

```
a = E
```

가 되죠. 따라서, a 에 E 의 값, 즉 b + c + d + e 의 값이 들어가게 됩니다.

또한, 위 표에서 몇 안되는 '오른쪽이 우선' 인 대입 연산자(=) 를 살펴봅시다. 만약 대입 연산자가 왼쪽 우선이였다면 아래의 식이 어떻게 계산될 지 생각해 봅시다.

```
a = b = c = d = 3;
```

만약 왼쪽 우선이였다면 a = b; b = c; c = d; d = 3 의 형식으로 계산되어 a, b, c 에는 알 수 없는 값이 들어가겠죠. 하지만 오른쪽이 우선이므로 위 식은 d = 3, c = d, b = c, a = b 의 형식으로 계산되어 a,b,c,d 의 값이 모두 3 이 될 수 있었습니다.

자, 이제 연산자에 대한 강의가 끝났습니다. 연산자는 C 언어에서 가장 기초적인 부분이라 할 수 있습니다. 마치 수학에서 덧셈, 뺄셈을 가장 처음에 배우는 것 처럼 말이죠.

이번 강좌에서는 특별히 예제를 많이 만들어 보지는 않았지만 여러분 께서 C 언어를 통해 복잡한 수식의 계산을 하거나, 복잡한 수식을 보고 이러한 연산은 이 순서로 연산될 것이다 라고 예측해 보는 것도 우선순위를 이해하는데 도움이 될 것입니다. 보통, 우선순위를 잘못 고려하여 나는 오류들은 찾기가 매우 힘들기 때문에 **애초에 헷갈릴 만한 부분은 괄호를 통해 확실하게 하는 것이 좋습니다**.

뭘 배웠지?

- +, -, /, *, =, % 가 무슨 역할을 하는 연산자 인지 배웠습니다.
- &, |, <<, >> 가 무슨 역할을 하는 연산자 인지 배웠습니다.
- 연산자 우선 순위에 대해 다루었습니다.
- 우선 순위가 헷갈리거나, 복잡한 수식이면 괄호를 적극 활용합니다.

컴퓨터가 음수를 표현하는 방법 (2의 보수)

안녕하세요 여러분! 지난 강좌에서 변수를 이용해서 여러가지 연산을 수행하는 방법에 대해 다루었습니다. 그런데 C 언어에서 아무런 제약 없이 연산을 수행할 수 있는 것은 아닙니다. 왜냐하면 변수마다 각각의 타입에 따라서 **보관할 수 있는 데이터의 크기** 가 정해져 있기 때문이죠.

예를 들어서 `int` 의 경우 -2147483648 부터 2147483647 까지의 정수 데이터를 보관할 수 있습니다.

그렇다면 아마 여러분들은 아래와 같은 질문을 하실 수 있습니다.

> 만약에 변수의 데이터가 주어진 범위를 넘어간다면 어떻게 되나요?

한 번 직접 코드를 작성해봅시다.

```c
#include <stdio.h>

int main() {
  int a = 2147483647;
  printf("a : %d \n", a);

  a++;
  printf("a : %d \n", a);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
a : 2147483647
a : -2147483648
```

와 같이 나옵니다.

```c
int a = 2147483647;
printf("a : %d \n", a);
```

먼저 위와 같이 `a` 라는 변수를 정의한 뒤에 `int` 가 표현할 수 있는 최대값인 2147483647 를 대입하였습니다. 해당 문장은 문제가 없으며 `printf` 로도 2147483647 가 잘 출력되었습니다.

```
a++;
printf("a : %d \n", a);
```

반면에 a++ 을 해서 int 가 표현할 수 있는 최대값을 넘어가버렸습니다. 그런데 놀랍게도 전혀 예상하지 못한 값이 출력되었습니다. 바로 **-2147483648** 이 나온 것이죠. 어떻게 양수에서 1 을 더했는데 음수가 나올 수 있을까요?

이에 대한 대답을 하기 위해선 먼저 컴퓨터에서 어떻게 음수를 표현하는지 알아야 합니다.

음수 표현 아이디어

여러분이 CPU 개발자라면 컴퓨터 상에서 정수 음수를 어떤식으로 표현하도록 만들었을까요? 가장 간단히 생각해보자면 우리가 부호를 통해서 음수 인지 양수 인지 나타내니까, 비슷한 방법으로 부호를 나타내기 위해서 1 비트를 사용하는 것입니다. (예를 들어서 0 이면 양수, 1 이면 음수) 그리고 나머지 부분을 실제 정수 데이터로 사용하면 되겠죠.

예를 들어서 가장 왼쪽 비트를 부호 비트라고 생각하자면 (참고로 아래 표현하는 수들은 모두 이진법으로 작성한 것입니다.)

```
0111
```

은 7 이 될 것이고

```
1111
```

은 맨 왼쪽 부호 비트가 1 이므로 -7 을 나타내게 됩니다. 꽤나 직관적인 방식이기는 하지만 여러가지 문제점이 있습니다. 첫 번째로 0 을 나타내는 방식이 두 개라는 점입니다. 즉

```
0000
```

도 0 이고

```
1000
```

역시 0 이지요. (-0 은 0 이니까) 뭐 *0* 이 두 개일 수 도 있지 라고 생각하실 수 있겠지만 사실 이는 매우 큰 문제 입니다. 왜냐하면 컴퓨터 상에서 어떠한 데이터가 0 인지 아닌지 비교하는 일을 굉장히 많이 하거든요.

0 을 표현하는 방법이 두 가지라면, 어떠한 데이터가 0 인지 확인하기 위해서 +0 인지 -0 인지 두 번이나 확인해야 하게 됩니다. 따라서 **이상한 데이터 표현법 덕분에 쓸데없이 컴퓨터 자원을 낭비하게 됩니다.**

또 다른 문제로는, 양수의 음수의 덧셈을 수행할 때 부호를 고려해서 수행해야 한다는 점입니다. 예를 들어서 `0001` 과 `0101` 을 더한다면 그냥 `0110` 이 되겠지만 `0001` 과 `1001` 을 더할 때에는 `1001` 이 사실은 `-1` 이므로 뺄셈을 수행해야 하죠. 따라서 덧셈 알고리즘이 좀 더 복잡해집니다.

물론 부호 비트를 도입해서 음수와 양수를 구분하는 아이디어 자체는 나쁜 생각은 아닙니다. 여기서는 `int` 와 같은 정수 데이터만 다루지만 `double` 이나 `float` 처럼 소수인 데이터를 다루는 방식에서는 (이를 부동 소수점 표현 이라고 하는데, 나중 강좌에서 자세히 알아봅시다.) 부호 비트를 도입하여서 음수인지 양수인지를 표현하고 있습니다.[3)]

하지만 적어도 정수를 표현하는 방식에서는 부호 비트를 사용하는 방식은 문제점이 있습니다.

2의 보수(2's complement) 표현법

그렇다면 다른 방법을 생각해봅시다. 만약에 어떤 x 와 해당 수의 음수 표현인 $-x$ 를 더하면 당연히도 0 이 나와야 합니다. 예를 들어서 7 을 이진수로 나타내면

```
0111
```

이 되는데 여기에 더해서 `0000` 이 되는 이진수가 있을까요? 이 때 덧셈 시에 컴퓨터가 4 비트만 기억한다고 가정합시다.

그렇다면 -7 의 이진수 표현으로 가장 적당한 수는 바로 `1001` 이 될 것입니다. 왜냐하면 `0111` 과 `1001` 을 더하면 `10000` 이 되는데, CPU 가 4 비트만 기억하므로 맨 앞에 1 은 버려져서 그냥 `0000` 이 되기 때문이지요.[4)]

이렇게 가장 덧셈을 고려하였을 때 가장 자연스러운 방법으로 음수를 표현하는 방식을 바로 2 의 보수 표현이라고 합니다. **2 의 보수 표현 체계 하에서 어떤 수의 부호를 바꾸려면 먼저 비트를 반전시킨 뒤에 1 을 더하면 됩니다.**

예를 들어서 -7 을 나타내기 위해서는, 7 의 이진수 표현인 `0111` 의 비트를 모두 반전시키면 `1000` 이 되는데 여기다 1 을 더해서 `1001` 로 표현하면 됩니다. 반대로 -7 에서 7 로 가고 싶다면 `1001` 의 부호를 모두 반전 시킨뒤 (0110) 다시 1 을 더하면 양수인 7 (0111) 이 나오게 됩니다.

이 체계에서 중요한 점은 `0000` 의 2 의 보수는 그대로 `0000` 이 된다는 점입니다. 왜냐하면 `0000` 을 반전하면 `1111` 이 되는데, 다시 1 을 더하면 `0000` 이 되기 때문이죠!

또한 어떤 수가 음수 인지 양수인지 판단하는 방법도 매우 쉽습니다. 그냥 맨 앞 비트가 부호 비트라고 생각하면 됩니다. 예를 들어서 `1101` 의 경우 맨 앞 비트가 1 이기 때문에 음수 입니다. 따라서 이 수가 어떤 값인지 알고싶다면 보수를 구한 뒤에 (1101 –> 0010 –> 0011) - 만 붙여주면 되겠죠. `0011` 이 3 이므로, `1101` 은 경우 -3 이 됩니다.

3) 실제로 부동 소수점 표현법에서는 -0 과 +0 이 있습니다.

4) 참고로 두 개의 자료형을 더했을 때 범위를 벗어나는 비트는 그냥 버려진다고 생각하시면 됩니다. 마치 왼쪽으로 쉬프트 했을 때 맨 왼쪽에 있는 비트들이 버려지는 것 처럼 말이죠.

이와 같이 2 의 보수 표현법을 통해서

- 음수나 양수 사이 덧셈 시에 굳이 부호를 고려하지 않고 덧셈을 수행해도 되고
- 맨 앞 비트를 사용해서 부호를 빠르게 알아낼 수 있다

와 같은 장점 때문에 컴퓨터에서 정수는 2 의 보수 표현법을 사용해서 나타내게 됩니다.

한 가지 재미있는 점은 2 의 보수 표현법에서 음수를 한 개더 표현할 수 있습니다. 왜냐하면 `1000` 의 경우 음수 이지만 변환 시켜도 다시 `1000` 이 나오기 때문이죠 (1000 –> 0111 –> 1000) 실제로 `int` 의 범위를 살펴보면 -2,147,483,648 부터 2,147,483,647 까지 이죠. 음수가 1 개 더 많습니다.

자 그렇다면 이전 코드를 다시 살펴봅시다.

```
int a = 2147483647;
printf("a : %d \n", a);

a++;
printf("a : %d \n", a);
```

처음에 a 에 `int` 최대값을 집어 넣었을 때 아마 a 에는 `0x7FFFFFFF` (이진수로 `0111 1111 ... 1111`) 라는 값이 들어가있을 것입니다. 그런데 여기서 1 을 더하게 되면 어떻게 될까요?

우리는 a 의 현재 값이 `int` 가 보관할 수 있는 최대값이므로 1을 더 증가 시킨다면 오류를 내뿜게하거나 아니면 그냥 현재 값 그대로 유지하게 하고 싶었을 것입니다.

하지만 CPU 는 그냥 `0x7FFFFFFF` 값을 1 증가 시킵니다. 따라서 해당 `a++` 이후에 a 에는 `0x80000000` (이진수로 `1000 0000 ... 0000`) 이 들어가겠죠. 문제는 `0x80000000` 을 2 의 보수 표현법 체계하에서 해석한다면 반전 하면 (`0111 1111 ... 1111`) 이 되고 다시 1 을 더하면 (`1000 0000 ... 0000`) 이 되므로 `-0x80000000`, 즉 -2147483648 이 됩니다.

따라서 위와 같이 양수에 1 을 더했더니 음수가 나와버리는 불상사게 생기게 되죠. 이와 같이 자료형의 최대 범위보다 큰 수를 대입하므로써 발생하는 문제를 **오버플로우(overflow)** 라고 하며, C 언어 차원에서 오버플로우가 발생하였다는 사실을 알려주는 방법은 없기 때문에 **여러분 스스로 항상 사용하는 자료형의 크기를 신경 써줘야만 합니다!**

주의 사항

실제로 1996년에 발사한 Arian 5 로켓은 제어 프로그램 상에서 발생한 오버플로우로 인해서 가속도를 제대로 계산하지 못해서 추락한 사례가 있습니다. 참고로 해당 로켓의 발사 비용은 3억 7천만 달러 (한화로 대략 4000 억원 정도 되죠) 였다고 합니다.

음수가 없는 자료형은 어떨까요?

unsigned int 의 경우 음수가 없고 0 부터 4294967295 까지의 수를 표현할 수 있습니다. unsigned int 가 양수만 표현한다고 해서 int 와 다르게 생겨먹은 것이 아닙니다. unsigned int 역시 int 와 같이 똑같이 32 비트를 차지 합니다.

다만, unsigned int 의 경우 int 였으면 2 의 보수 표현을 통해 음수로 해석될 수를 그냥 양수라고 생각할 뿐이지요.

따라서 unsigned int 에 예를 들어서 -1 을 대입하게 되면, -1 은 0xFFFFFFFF 로 표현되니까,

```c
#include <stdio.h>

int main() {
  unsigned int b = -1;
  printf("b 에 들어있는 값을 unsigned int 로 해석했을 때 값 : %u \n", b);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
b 에 들어있는 값을 unsigned int 로 해석했을 때 값 : 4294967295
```

와 같이 나옵니다. 참고로 printf 에서 %u 는 unsigned 타입으로 해석하라는 의미 입니다.

물론 unsigned int 상에서도 오버플로우가 발생하지 않으라는 법이라는 없습니다. 예를 들어서 b 에 최대값을 대입한 뒤에 1 을 추가한다면;

```c
#include <stdio.h>

int main() {
  unsigned int b = 4294967295;
  printf("b : %u \n", b);

  b++;
  printf("b : %u \n", b);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
b : 4294967295
b : 0
```

와 같이 나옵니다. 즉, b 에 0xFFFFFFFF (1111 1111 ... 1111) 이 들어가 있다가 1 증가함으로써 (1 0000 ... 0000) 이 되었는데 앞서 이야기 하였듯이 자료형의 크기를 초과하는 비트는 그대로 버려지므로 그냥 0 (0000 0000 ... 0000) 으로 해석된 것입니다.

즉 unsigned int 역시, 아니 C 언어 상에 모든 자료형은 오버플로우의 위험으로 부터 자유롭지 않습니다.

자 그럼 이것으로 이번 강좌를 마치도록 하겠습니다. C 언어에서 코딩을 할 때에는 언제나 오버플로우 문제에 신경써야 합니다. 또한 아니 *int* 에 분명히 양수만 더했는데 왜 음수가 나왔지*?* 와 같은 상황에 당황하지 않고 대처할 수 있겠죠!

뭘 배웠지?

- 컴퓨터 상에서 정수인 음수를 표현하기 위해서 2 의 보수 표현법을 사용합니다.
- 이에 따라 int 상에서 오버플로우가 발생하였을 때 양수에서 값을 증가시켰더니 음수로 바뀌는 기적을 볼 수 있습니다.
- 항상 오버플로우를 조심합시다.

4

문자 입력 받기

지난번 강좌는 잘 이해 되셨는지요? 이번 강좌에서는 제목에서도 볼 수 있듯이 두 가지 내용을 한꺼번에 배우게 됩니다. 바로, 문자를 키보드로 부터 입력을 받는 것이지요. 문자를 입력 받을 수 있다면, 숫자도 당연히 입력 받을 수 있게 됩니다. 즉, 이번 강좌에서는 문자 형식의 변수와 키보드로부터 입력을 받는 입력에 대해 알아 보도록 하겠습니다.

일단, 컴퓨터에서 문자를 처리하는 방식에 대해 생각해 봅시다. 제가 누누히 말하지만 우리의 컴퓨터는 그다지 똑똑하지 못합니다. 아무리 최신 `Intel CPU` 를 장착해도 컴퓨터는 단지 0 과 1 만을 처리할 뿐이죠.

따라서, 2 와 3 같은 숫자도 처리하지 못하는데 어떻게 a, b 가, 나, 韓 과 같은 수 많은 문자를 처리할 수 있겠습니까? 하지만, 방법이 있습니다. 이러한 문자들을 숫자에 대응시키는 것입니다. 그런데, 숫자에 대응시킨다면 컴퓨터가 이 것이 숫자인지, 아니면 문자인지 어떻게 알까요? 물론 알 방법은 없습니다. 단지 이 숫자를 '문자' 형태로 사용하거나 '숫자' 형태로 사용하는 것이지요.

문자를 저장하는 변수는 앞에서 살짝 본 적이 있습니다. 바로 `char` 이지요. `int` 가 `integer` 의 약자였다면 `char` 은 `character` 의 약자 입니다. 변수가 등장하면 어김없이 등장하는 아래의 표를 살펴 봅시다.

Name	Size*	Range*
char	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	1byte	true or false
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

보시는 것과 같이 `char` 은 맨 위에 위치해 있으며 크기는 1 바이트 입니다. 또한, 이를 통해 나타낼

수 있는 숫자의 범위를 알려주고 있는데, 이는 -128 부터 127 까지, 256 가지 입니다.

```c
/* 문자를 저장하는 변수 */
#include <stdio.h>
int main() {
  char a;
  a = 'a';

  printf("a 의 값과 들어 있는 문자는? 값 : %d , 문자 : %c \n", a, a);
  return 0;
}
```

위 소스를 성공적으로 컴파일 했다면

실행 결과

```
a 의 값과 들어 있는 문자는? 값 : 97 , 문자 : a
```

위와 같이 나옵니다. 일단, 소스를 분석해 보겠습니다.

```c
char a;
```

이 부분은 char 형 변수를 선언하는 부분입니다. 기억이 안나시는 분들은 3강을 참조하세요.

```c
a = 'a';
```

이 부분은 a 라는 변수에 문자 a 를 대입하고 있습니다. 이 때, 모든 문자들은 모두 작은 따옴표로 묶어 주어야 합니다. 만약 작은 따옴표로 묶지 않고 그냥 썼다면

```c
a = a;
```

C 컴파일러는 이 a 가 변수 a 라고 착각하여 a 라는 변수의 값을 a 라는 변수에 대입하는 문장으로 인식하게 되죠. 따라서 a 에는 아무런 값이 들어있지 않은 쓰레기 값(NULL) 이 되어 나중에 a 라는 문자를 출력해 보았을 때, 이상한 값이 나오게 됩니다.

문자를 대입하는 것도 숫자를 대입하는 것과 동일합니다. 대입 연산자를 이용하면 되죠.

```c
printf("a 의 값과 들어 있는 문자는? 값 : %d , 문자 : %c \n", a, a);
```

마지막으로, 위 `printf` 문에 대해 보도록 하겠습니다. 앞에서 말했듯이 컴퓨터는 a 가 문자라는 것 자체를 모른다고 했습니다. 단지 우리가 a 를 문자로 보느냐 아니면 숫자를 보느냐에 따라 달라진다고 했는데, 이 말 뜻을 위 `printf` 문을 보면 알 수 있습니다.

일단, %d 는 a 의 값을 숫자 (정수인 10 진수) 라고 출력하라는 뜻입니다. 그 옆의 %c 는 아마 예상했겠지만 a 의 값을 문자로 출력하라는 뜻이지요. 따라서, %c 에는 a 에 저장되어 있던 문자 'a' 가 출력되게 됩니다.

그렇다면 %d 에는 무엇이 출력되었을까요? 앞에서 말했지만 컴퓨터는 문자와 숫자를 일대일 대응시켜서 생각한다고 했습니다. 따라서, %d 에 출력되는 숫자가 바로 a 에 대응되는 숫자를 가리킵니다.

이 때, 각 문자 마다 대응되는 숫자를 아무렇게나 하는 것이 아니라 일정하게 정해져 있는데 현재 우리가 쓰고 있는 컴퓨터에서는 다음과 같이 정의되어 있습니다.

10진수	ASCII	10진수	ASCII	10진수	ASCII	10진수	ASCII
0	NULL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	`	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	`	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	SC2	50	2	82	R	114	r
19	SC3	51	3	83	S	115	s
20	SC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	₩	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

위 표는 미국 표준 학회(ASA) 에서 정한 **아스키**(ASCII, *American Standard Code for Information Interchange*) 코드로 8 비트 데이타를 이용하여 여러 문자에 번호를 붙인 것 입니다. 아까, a 의 숫자 값을 출력하였을 때 97 이 나왔는데 위 표에서 찾아 보면 a 의 값이 97 임을 볼 수 있습니다. 이 때, 위 표의 내용이 0 부터 127 까지 밖에 없는 이유는 위 표준을 정할 당시 그 당시 7 비트 만으로 충분하다고 생각했기 때문이죠. 하지만 IBM 에서 좀 더 많은 종류의 문자가 필요하게 되자 1 비트를 더 추가 시켜서 확장된 아스키 코드(Extended ASCII Code) 를 만들었습니다.

하지만 위 256 개 가지고는 충분하지 못하죠. 왜냐하면 우리 글만 해도 자모음 24 개로 구성되어 있는데, 한 글자당 최대 초성/중성/종성 을 모두 표현해야 합니다. 또한 더욱 심각한 것은 한자와 같은 표의문자의 경우 수만 개가 넘는 한자 데이터들을 가지고 있어야 하는데 이를 256 개 안에 다 표현한다는 것은 불가능하기 때문이죠. 따라서, 컴퓨터가 전세계에 보급되자 좀 더 많은 종류의 문자를 표현해야 한다는 필요성이 대두되었습니다.

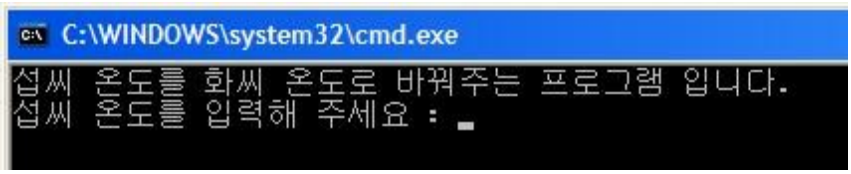
결국에는 **유니 코드(Unicode)** 라는 새로운 형식의 문자 체계를 도입하게 됩니다. 유니코드는 한 문자를 1 에서 4 바이트까지 다양한 길이로 처리합니다. 이는, 기존 아스키 코드의 체계를 유지하면서, 새로운 문자들을 추가하기 위함입니다. 여러분은 아직 유니코드를 직접 다룰 일은 없기 때문에 여기서는 무시하셔도 괜찮습니다.

scanf 의 도입

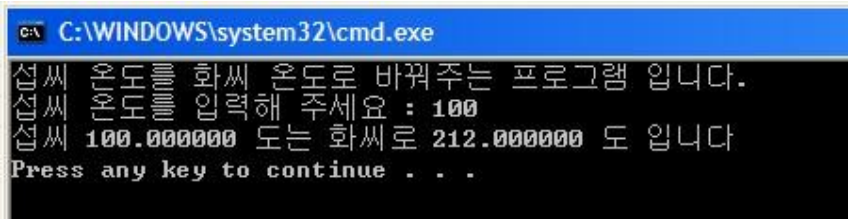
```c
/* 섭씨온도를 화씨로 바꾸기 */
#include <stdio.h>
int main() {
  double celsius;  // 섭씨 온도

  printf("섭씨 온도를 화씨 온도로 바꿔주는 프로그램 입니다. \n");
  printf("섭씨 온도를 입력해 주세요 : ");
  scanf("%lf", &celsius);  // 섭씨 온도를 입력 받는다.

  printf("섭씨 %f 도는 화씨로 %f 도 입니다 \n", celsius, 9 * celsius / 5 + 32);

  return 0;
}
```

위 소스를 성공적으로 컴파일 했다면 아래와 같이 나옵니다. 참고로 scanf_s 를 사용하라며 컴파일 되지 않는다면 여기 에 소개된 방법으로 해당 메세지를 끌 수 있습니다.[1)]

1) scanf_s 를 사용하라는 의미는 scanf 가 입력받는 데이터의 크기를 확인하지 않기 때문에 버퍼 오버플로우 (입력받는 데이터의 크기가 준비된 공간보다 큰 문제) 가 발생할 수 있기 때문입니다. 이 문제에 대해서는 나중에 다루어보도록 하겠습니다.



이 때, 원하는 숫자를 쓴 후 엔터를 누른다면 (예 : 100)


C:\WINDOWS\system32\cmd.exe
섭씨 온도를 화씨 온도로 바꿔주는 프로그램 입니다.
섭씨 온도를 입력해 주세요 : 100
섭씨 100.000000 도는 화씨로 212.000000 도 입니다
Press any key to continue . . .

위와 같이 섭씨가 화씨 온도로 변경된 값이 출력됩니다. 와우! 드디어 쓸만한 프로그램을 처음으로 만들어 보게 된 것 같군요. 소스 코드를 찬찬히 살펴 보도록 합시다.

```
double celsius;  // 섭씨 온도
```

일단, celsius 라는 double 형 변수를 선언하였습니다. 변수의 이름을 종전의 a , b 에서 celsius 라고 한 이유는 좀 더 이해하기 편하기 때문이죠. 좋은 소스 코드의 조건은 다른 사람이 이해하기 쉬운 소스 코드 이고, 다른 사람이 이해하기 쉬운 소스코드는 기본적으로 변수 이름을 보고도 변수를 한 눈에 파악하기 쉽게 만드는 것입니다.

```
scanf("%lf", &celsius);  // 섭씨 온도를 입력 받는다.
```

이제, 새로운 것이 등장하였군요. printf 에 이어 등장한 scanf 군. printf 가 화면에 결과를 출력해 주는 함수였다면, scanf 는 화면(키보드) 로 부터 결과를 받아들이는 입력 함수 입니다. 이렇게 흔히 printf 와 scanf 를 가리켜 입출력함수라 하죠. 이 때, scanf 함수는 우리가 어떠한 입력을 하기 전까지 계속 기다립니다. 또한, 입력을 할 때 엔터를 눌러야지만 입력으로 처리됩니다.

scanf 와 printf 는 이름도 비수무리 할 뿐더러, 사용하는 방법도 비슷합니다. printf 에서 각 변수를 출력할 포맷(%d, %f, %c 등) 을 변수마다 다르게 하는 것처럼 scanf 도 각 변수의 타입마다 입력받는 포맷을 달리 해야 합니다.

위 경우 처럼 double 형의 변수를 입력 받으려면 %lf (소문자 LF 이다, if 가 아니다) 로 해야 합니다. 그런데, printf 보다 조금 까다로운 점은 printf 는 double 이나 float 모두 %f 로 출력하지만 이에 경우 float 은 %f 로 무조건 입력 받아야 한다는 점입니다.

마찬가지로 double 형 변수도 무조건 %lf 로만 입력 받아야 합니다. 그 외에도, printf 는 정수형 변수는 모두 %d 로 출력 가능했던 반면에 scanf 는 각 자료형 마다 포맷이 다 정해져 있습니다. 아래 예제에서 잠시 scanf 의 포맷 들에 대해 정리해 보도록 하겠습니다

```
printf("섭씨 %f 도는 화씨로 %f 도 입니다 \n", celsius, 9 * celsius / 5 + 32);
```

마지막으로 위 프로그램의 중요한 부분을 살펴보자. 바로 이 부분에서 섭씨와 화씨의 환산 작업이 이루어 진다. 참고로, 화씨와 섭씨의 변환 공식은 아래와 같습니다.

$$C \cdot \frac{9}{5} + 32 = F$$

따라서, 이 공식을 그대로 C 언어 수식을 바꾼 것이 `9 * celsius / 5 + 32` 인 것입니다. 곱셈과 나눗셈의 우선순위가 높으므로 `9 * celsius / 5` 가 먼저 계산 된 후 32 가 더해지므로 위의 식과 일치합니다. 따라서 `printf` 의 두번째 `%f` 부분에는 위 계산된 화씨의 값이 들어가게 됩니다.

```c
/* scanf 총 정리 */
#include <stdio.h>
int main() {
  char ch;  // 문자

  short sh;  // 정수
  int i;
  long lo;

  float fl;  // 실수
  double du;

  printf("char 형 변수 입력 : ");
  scanf("%c", &ch);

  printf("short 형 변수 입력 : ");
  scanf("%hd", &sh);
  printf("int 형 변수 입력 : ");
  scanf("%d", &i);
  printf("long 형 변수 입력 : ");
  scanf("%ld", &lo);

  printf("float 형 변수 입력 : ");
  scanf("%f", &fl);
  printf("double 형 변수 입력 : ");
  scanf("%lf", &du);

  printf("char : %c , short : %d , int : %d ", ch, sh, i);
  printf("long : %ld , float : %f, double : %f \n", lo, fl, du);
  return 0;
}
```

성공적으로 컴파일 후 (경고가 6 개 정도 나올 수 있는데 무시하세요)

```
printf("char 형 변수 입력 : ");
scanf("%c", &ch);
```

일단, 제일 먼저 문자를 입력 받는 부분을 봅시다. 예전에도 이야기 했지만 한글은 2 바이트 이상을 차지하기 때문에 최대 1 바이트를 차지하는 char 형 변수인 ch 에 한글을 치면 오류가 납니다. 이와 같이 허용된 메모리 이상에 데이터를 집어넣어 발생하는 오류를 **버퍼 오버플로우(Buffer Overflow)** 라고 하며 보안 상 매우 취약합니다.[2] 뿐만 아니라 근처의 데이터가 손상됨에 따라 큰 문제가 발생하게 될 수 도 있습니다. 따라서, 여러분들은 버퍼 오버플로우가 일어나지 않게 허용된 데이터 이상을 집어넣는지 안집어 넣는지 검사해야 합니다.

또한 앞으로 우리가 char 형 변수를 선언할 때 에는 이 사람이 문자를 보관하는 변수를 선언하는 구나 라고 생각하도록 합시다. 왜냐하면 보통 정수 데이터를 보관하는 변수로는 int 를 쓰지 char 을 잘 쓰지 않을 뿐더러 char 이름도 character 에서 따왔을 만큼 문자와 무언가 관련이 있기 때문이죠.

```
printf("short 형 변수 입력 : ");
scanf("%hd", &sh);
printf("int 형 변수 입력 : ");
scanf("%d", &i);
printf("long 형 변수 입력 : ");
scanf("%ld", &lo);
```

이 부분은 여러분들이 무난하게 이해하실 수 있으리라 봅니다. 단지 포맷에 %hd, %d, %ld 로 다른 것 뿐이지요. 참고로 short 형이나 long 형은 아직 다루지는 않았지만 int 와 똑같은 계열의 정수형 변수라고 생각하시면 됩니다.

```
printf("float 형 변수 입력 : ");
scanf("%f", &fl);
printf("double 형 변수 입력 : ");
scanf("%lf", &du);
```

마찬가지로 float 형에서는 %f 로, double 형에서는 %lf 로 사용한다는 것을 기억하시기 바랍니다.

2) 버퍼 오버플로우를 이용해서 공격자들이 프로그램의 정상적인 코드가 아니라, 자신들이 원하는 코드가 실행될 수 있도록 조종할 수 있습니다.

이번 강좌는 지난번 강좌보다는 조금 짧습니다. 하지만 이번 강좌를 통해 응용할 수 있는 것들이 무궁무진해졌습니다. 일단, 기본적으로 연습하실 것은 단위 환산 프로그램을 만들어 보세요! 아니면, 금리와 원금을 입력 받아서 일정 개월 후의 상환할 돈이라 든지 등등... 수 많은 프로그램을 만들 수 있습니다. 지금, 이러한 것들을 만들 수 있는 모든 도구들은 준비되어 있습니다. 이제 여러분이 스스로 창작할 세계가 남아 있을 뿐입니다.

생각 해보기

문제 1

앞서 섭씨를 화씨로 바꿀 때 `9 * celsius / 5 + 32` 라고 하였습니다. 만약에 이를 `9 / 5 * celsius + 32` 로 바꾸면 결과가 달라질까요?

뭘 배웠지?

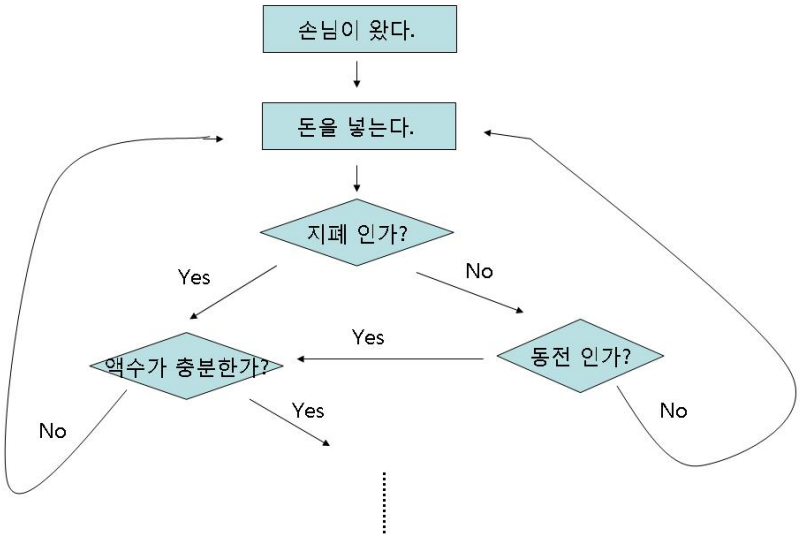
- `char` 은 1 바이트 정수를 저장하는 타입으로, 주로 문자를 저장하는데 사용됩니다.
- 각 문자들은 아스키 테이블이란 표를 통해 특정 정수와 대응되어 있습니다. 예를 들어서 65 는 알파벳 A 와 대응됩니다.
- `scanf` 를 통해 사용자로 부터 데이터를 받을 수 있습니다.
- `%c` 는 문자, `%d` 는 정수, `%f` 는 `float`, `%lf` 는 `double` 을 받습니다.

5

만약에... (조건문)

안녕하세요? 여러분. 이제 저의 강좌도 6강에 이르렀네요. 지난번의 강좌에서 배운 입출력 함수로 여러가지 재미있는 프로그램을 만들었나요? 그러한 프로그램들을 많이 만들 수록 여러분의 실력은 몇 배로 향상된다는 사실을 잊지 마시기 바랍니다. 이번 강좌에는 C 언어에서 매우 중요한 부분인 제어문 - 그 중에서도 조건문이라는 사실을 배우겠습니다.

우리가 자판기에서 음료수를 고를 때 자판기 내부에는 다음과 같은 과정이 수행 됩니다.



돈을 넣는다. 지폐인가? 맞다면 액수가 충분한가? 아니면 더 받는다 .... 이렇게 맞다 아니다 로 갈리게 됩니다. 이러한 것을 조건문 이라고 합니다. 즉 항상 실행되는 것이 아니라 특정한 조건이 맞는 경우에만 실행되는 것이지요

그런데, 위 부분에서 우리가 여태 까지 보지 못했던 특징이 있습니다. 여태까지 우리는 모든 문장들이 순차적으로 실행되어 왔습니다. 예를들어,

```
//.....
```

```
printf("안녕\n");
printf("내 이름은 Psi 야 \n");
printf("너의 이름은 뭐니?\n");
//....
```

와 같은 문장에서 처음에 **안녕** 이 출력되고 그 다음에 **내 이름은 Psi 야**, 그리고 마지막으로 **너의 이름은 뭐니?** 가 출력이 됩니다. 안녕이 출력이 되지 않거나 너의 이름은 뭐니? 가 출력되지 않는 일은 없죠.

뿐만 아니라, 내 이름은 Psi 야가 안녕 보다 먼저 출력되는 경우도 없습니다. 단지 어떠한 조건에서도 안녕, 내 이름은 Psi 야, 너의 이름은 뭐니? 가 차례대로 출력되는 것이지요.

하지만, 위의 그림을 봅시다. **지폐 인가?** 부분을 보면 만약 Yes 라면 **액수는 충분한가?** 를 실행하고 No 라면 **동전 인가?** 를 실행하게 끔 되어 있습니다. 다시말해 어떠한 경우를 만족한다면 이것을 실행하고, 또 다른 경우라면 이 것을 실행하는 꼴이지요. 이런 것을 '조건문' 이라 합니다. 어떠한 조건에 따라 실행되는 것이 달라지는 것이지요.

if 문 시작하기

아래 예제를 봅시다.

```
/* if 문 이란? */
#include <stdio.h>
int main() {
  int i;
  printf("입력하고 싶은 숫자를 입력하세요! : ");
  scanf("%d", &i);

  if (i == 7) {
    printf("당신은 행운의 숫자 7 을 입력했습니다");
  }

  return 0;
}
```

위 예제를 성공적으로 컴파일 한 후, 7 을 입력하였으면

실행 결과

```
입력하고 싶은 숫자를 입력하세요! : 7
당신은 행운의 숫자 7 을 입력했습니다
```

와 같이 나오게 됩니다. 그런데, 7 이 아닌 다른 수를 입력하였을 때 에는,

실행 결과

```
입력하고 싶은 숫자를 입력하세요! : 1
```

처럼 나오게 됩니다. 일단, 위 소스 코드에는 우리가 여태까지 보지 못했던 것이 있으니 찬찬히 뜯어보도록 합시다.

```
if (i == 7) {
  printf("당신은 행운의 숫자 7 을 입력했습니다");
}
```

이는위 소스코드에서 가장 핵심적인 부분입니다. 영어에서 흔히 말할 때 어떨 때, if 라는 단어를 쓰나요. 예를 들어서, *If you are smart enough, please learn C language* (만약 당신이 충분히 똑똑하다면, C 언어를 배워라!) 라는 문장을 보았을 때 **If** 라는 단어는 무슨 역할을 하나요? 아마 영어를 조금이나마 배운 사람들이라면, **if** 는 '만약 ~ ' 이라는 의미를 가진 것임을 바로 알 수 있습니다.

C 언어에서도 마찬가지 입니다. `if` 는 가정을 나타냅니다. 여기서는 무엇을 가정 하였을까요? 바로 `if` 문 안에 있는 `i == 7` 이 그 가정을 나타냅니다. 즉, 만약 i 의 값이 7 이라면 이라는 뜻이지요.

`if` 문은 언제나 괄호 안의 조건이 참 이라면 중괄호 속의 내용을 실행하게 되고, 아니면 중괄호 속의 내용을 실행하지 않고 지나치죠.

따라서, 만약 `i` 의 값이 7 이라면,

```
printf("당신은 행운의 숫자 7 을 입력했습니다");
```

를 실행 한 후 중괄호 밖으로 빠져 나갑니다. 그 후 다시 아래로 순차적으로 실행하게 되죠. 즉, 마지막으로 `return 0;` 가 실행됩니다.

그런데, `i` 의 값이 7 이 아니라면, `if` 문의 중괄호 속의 내용은 실행되지 않고 지나쳐 버립니다. 왜냐하면 `if` 문 에서 `i == 7` 이 '거짓' 이 되기 때문이죠. 따라서 그냥 `return 0` 만 실행이 됩니다.

참고적으로 == 와 같이 어떠한 두 값 사이의 관계를 나타내 주는 연산자를 **관계 연산자** 라고 부릅니다. 이 때, 관계 연산자의 좌측에 있는 부분을 좌변, 우측에 있는 부분을 우변이라 합니다. (즉, `3 == 2` 와 같은 경우 3 은 좌변, 2 는 우변 이라 부릅니다)

또한, 알아야 될 또 한가지 중요한 것은, 사실 관계 연산자는 어떠한 관계를 연산 한 후에, 참 이면 1 을, 거짓이면 0 을 나타내게 됩니다.

다시 말해, `if` 문은 참, 거짓에 따라서 중괄호 속의 내용을 실행 하느냐, 하지 않느냐 결정하는 것 처럼 보이지만 실제로는, **`if` 문 속의 조건이 0 인가 (거짓), 0 이 아닌가 (참) 에 따라서 실행의 유무를 판별하게 되죠**

따라서, `if (0)` 이라 한다면 그 중괄호 속의 내용은 절대로 실행되지 않고, `if(1)` 이라 한다면 그 중괄호 속의 내용은 100% 실행되게 됩니다.

```c
/* 나눗셈 예제 */
#include <stdio.h>
int main() {
  double i, j;
  printf("나누고 싶은 두 정수를 입력하세요 : ");
  scanf("%lf %lf", &i, &j);

  printf("%f 를 %f 로 나눈 결과는 : %f \n", i, j, i / j);

  return 0;
}
```

성공적으로 컴파일 후 10 과 3 을 입력하였다면 아래와 같이 나오게 됩니다

실행 결과

```
나누고 싶은 두 정수를 입력하세요 : 10
3
10.000000 를 3.000000 로 나눈 결과는 : 3.333333
```

와우! 성공적으로 되었습니다. 그런데 위에서 나온 소스 코드는 우리가 여태까지 바왔던 코드와 다를 바가 없습니다. 여태까지 배운 기능들만 이용해서 만든 것이지요. 하지만 바로 이 부분에서 문제가 대두됩니다.

컴파일한 프로그램을 다시 실행시켜 1 과 0 을 차례대로 입력해 보세요. 즉, 1 을 0 으로 나누어 보세요.

실행 결과

```
나누고 싶은 두 정수를 입력하세요 : 1
0
1.000000 를 0.000000 로 나눈 결과는 : inf
```

위에서 보시는 것과 같이 나눈 결과가 `inf` 이라는 이상한 결과를 내뿜었습니다. 왜 일까요? 왜냐하면 수학에서, (즉 컴퓨터 에서) 어떠한 수를 0 으로 나누는 것은 금지되어 있기 때문이죠. 위 `i` 와 `j` 변수가 `double` 로 선언되어 있어서 망정이지 `i,j` 변수를 `int` 형으로 선언하였다면 프로그램은 애러를 내뿜고 종료 됩니다.

이 문제를 대수롭지 않게 여긴다면 큰 문제입니다. 예를들어서 여러분이 엑셀로 열심히 작업을 하였는데 실수로 어떤 수를 0 으로 나누는 작업을 하였더니 힘들게 작업한 엑셀이 종료되 버려 파일이

날아가 버리면 여러분은 다시는 엑셀을 쓰지 않을 것 입니다. 따라서, 우리는 나누는 수 (제수) 가 0 이 아닌지 확인할 필요성이 있습니다. 즉, 제수가 0 이면 나누지 않고 0 이 아니면 나누는 것이지요.

따라서 위 프로그램을 아래와 같이 수정합시다.

```c
#include <stdio.h>
int main() {
  double i, j;
  printf("나누고 싶은 두 정수를 입력하세요 : ");
  scanf("%lf %lf", &i, &j);

  if (j == 0) {
    printf("0 으로 나눌 수 없습니다. \n");
    return 0;
  }
  printf("%f 를 %f 로 나눈 결과는 : %f \n", i, j, i / j);

  return 0;
}
```

만약 1 을 0 으로 나누었다면

실행 결과

```
나누고 싶은 두 정수를 입력하세요 : 1
0
0 으로 나눌 수 없습니다.
```

그리고 다시 10 을 3 으로 나누어 보면

실행 결과

```
나누고 싶은 두 정수를 입력하세요 : 10
3
10.000000 를 3.000000 로 나눈 결과는 : 3.333333
```

로 위와 같이 정상적으로 나타납니다. 그럼 위 소스코드를 뜯어 보기로 합시다.

```c
{
  printf("0 으로 나눌 수 없습니다. \n");
  return 0;
}
```

만약 j 의 값이 0 이라면 중괄호 속의 내용이 실행되며, *0* 으로 나눌 수 없습니다 가 표시되고 프로그램이 종료(return 0) 됩니다.

반면에, j 의 값이 0 이 아니라면 중괄호 속의 내용이 실행되지 않습니다. 즉, 아래의 내용이 실행되게 됩니다.

```
printf("%f 를 %f 로 나눈 결과는 : %f \n", i, j, i / j);

return 0;
```

이렇듯, if 문은 여러 조건에 따른 처리를 위해 사용합니다. 먼저 나왔던 예제는 i 의 값이 7 일 때의 처리를 위해 if 문을 사용하였고 위의 예제는 j 의 값이 0 일 때의 처리를 위해 사용하였습니다.

```
/* 합격? 불합격? */
#include <stdio.h>
int main() {
  int score;

  printf("당신의 수학점수를 입력하세요! : ");
  scanf("%d", &score);

  if (score >= 90) {
    printf("당신은 합격입니다! \n");
  }

  if (score < 90) {
    printf("당신은 불합격 입니다! \n");
  }

  return 0;
}
```

위 소스를 성공적으로 컴파일하였다면 다음과 같이 나옵니다.만약 당신의 수학점수로 91 점을 입력하였다면,

실행 결과

```
당신의 수학점수를 입력하세요! : 91
당신은 합격입니다!
```

만약 당신의 수학 점수로 80 점을 입력하였다면

실행 결과

```
당신의 수학점수를 입력하세요! : 80
당신은 불합격 입니다!
```

와 같이 나타납니다.

```c
if (score >= 90) {
  printf("당신은 합격입니다! \n");
}

if (score < 90) {
  printf("당신은 불합격 입니다! \n");
}
```

위 소스의 핵심이라 할 수 있는 부분은 위 두 부분입니다.일단, `if(score >= 90)` 이라는 부분부터 살펴 봅시다. 이미 짐작은 했겠지만 `>=` 은 ~ 이상, 즉 ~ 보다 크거나 같다 를 의미합니다. 따라서, `score` 의 값이 90 보다 '크거나 같으면' `if` 문 안의 내용이 참 (true) 이 되어 중괄호 속의 내용이 실행됩니다.

따라서, 처음에 우리가 91 점을 입력하였을 때 `score >= 90` 이 참이 되어서

```c
printf("당신은 합격 입니다! \n");
```

가 실행되었습니다. 그런데 여기서 주의해야 할 점은 `score => 90` 이라고 하면 안된다는 것입니다. 이렇게 쓰면 컴파일러는 인식을 하지 못합니다.

이 부분에서 헷갈리는 사람들은 '크거나 같다' 라는 말 그대로 옮겨 적었다고 생각하세요. `>=` 는 크거나 (>) 같다 (=) 를 합친 것이다!

마찬가지로, 아래 `score < 90` 도 보자면 `score` 가 90 미만일 때 참이다 라는 사실을 나타낸 것임을 알 수 있습니다.

```c
/* 크기 비교하기 */
#include <stdio.h>
int main() {
  int i, j;

  printf("크기를 비교할 두 수를 입력해 주세요 : ");
  scanf("%d %d", &i, &j);

  if (i > j)  // i 가 j 보다 크면
  {
    printf("%d 는 %d 보다 큽니다 \n", i, j);
```

```
  }
  if (i < j)  // i 가 j 보다 작으면
  {
    printf("%d 는 %d 보다 작습니다 \n", i, j);
  }
  if (i >= j)  // i 가 j 보다 크거나 같으면
  {
    printf("%d 는 %d 보다 크거나 같습니다 \n", i, j);
  }
  if (i <= j)  // i 가 j 보다 작거나 같으면
  {
    printf("%d 는 %d 보다 작거나 같습니다 \n", i, j);
  }
  if (i == j)  // i 와 j 가 같으면
  {
    printf("%d 는 %d 와(과) 같습니다 \n", i, j);
  }
  if (i != j)  // i 와 j 가 다르면
  {
    printf("%d 는 %d 와(과) 다릅니다 \n", i, j);
  }

  return 0;
}
```

위 내용을 성공적으로 컴파일 후, 10 과 4 를 입력하였다면

실행 결과

```
크기를 비교할 두 수를 입력해 주세요 : 10 4
10 는 4 보다 큽니다
10 는 4 보다 크거나 같습니다
10 는 4 와(과) 다릅니다
```

와 같이 나타나게 됩니다.

이번 예제에서는 소스의 길이가 약간 깁니다. 하지만 따지고 보면 상당히 간단한 구조로 되어 있음을 알 수 있습니다. 일단 위의 예제에 관계 연산자 들의 역할에 대해 알아봅시다.

1. `>=` : 좌변이 우변보다 같거나 크면 참이 됩니다. (`6 >= 3` : 참, `6 >= 6` : 참, `6 >= 8` : 거짓)
2. `>` : 좌변이 우변보다 크면 참이 됩니다. (`6 > 3` : 참, `6 > 6` : 거짓, `6 > 8` : 거짓)
3. `<=` : 좌변이 우변보다 작거나 같으면 참이 됩니다. (`6 <= 3` : 거짓, `6 <= 6` : 참, `6 <= 8` : 참)

4. `<` : 좌변이 우변보다 작으면 참이 됩니다.(`6 < 3` : 거짓, `6 < 6` : 거짓, `6 < 8` : 참)
5. `==` : 좌변과 우변이 같으면 참이 됩니다.(`6 == 3` : 거짓, `6 == 6` : 참, `6 == 8` : 거짓)
6. `!=` : 좌변과 우변이 다르면 참이 됩니다. (`6 != 3` : 참, `6 != 6` : 거짓, `6 != 8` : 참)

따라서, 위 관계연산자에 따라 위 프로그램이 실행이 됩니다. 한 번 여러가지 숫자를 집어 넣으면서 확인해 보세요.

마지막으로 `if` 문의 구조에 대해서 간단히 정리해 보자면

```
if (/* 조건 */) {
  /* 명령 */
}
```

와 같이 됩니다. 잊지 마세요!

if - else 문 시작하기

```
#include <stdio.h>
int main() {
  int num;

  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("행운의 숫자 7 이군요!\n");
  } else {
    printf("그냥 보통 숫자인 %d 를 입력했군요\n", num);
  }
  return 0;
}
```

만약 성공적으로 컴파일 하였다면 7 을 입력했을 때,

실행 결과

```
아무 숫자나 입력해 보세요 : 7
행운의 숫자 7 이군요!
```

그리고, 그 외 7 이 아닌 다른 숫자를 입력하였을 때 에는,

실행 결과

```
아무 숫자나 입력해 보세요 : 100
그냥 보통 숫자인 100 를 입력했군요
```

와 같이 나오게 됩니다. 자, 이제 위 소스를 뜯어 봅시다.

```
if (num == 7) {
  printf("행운의 숫자 7 이군요!\n");
}
```

여태 까지 보왔듯이, 이 부분은 그냥 평범한 `if` 문이군요. 하지만 그 다음 부분에 심상치 않은 것이 등장합니다.

```
else {
  printf("그냥 보통 숫자인 %d 를 입력했군요\n", num);
}
```

이번에는 여태까지 배우지 않은 것인 `else` 라는 것이 등장합니다. 영어를 잘 하시는 분들은 지레 짐작하시고 있었겠지만 `else` 는 '그 외의~ , 그 밖의~' 의 뜻으로 사용되는 단어입니다.

그렇다면 여기서도 그러한 의미를 나타내는 것인가요?

맞습니다. `else` 는 바로 '앞선 `if` 문이 조건을 만족하지 않을 때' 를 나타냅니다. 즉, 앞선 `if` 문이 조건을 만족 안할 때 해야 할 명령을 바로 `else` 문에 써 주는 것이지요. 다시 말해, `else` 문은 떨거지(?) 들을 처리하는 부분입니다.

위의 경우에서도 만약 `num` 의 값이 7 이 아니였다면 `if` 문을 만족 안하게 되는 것입니다. 그러면 자연스럽게 `else` 로 넘어와서 "그냥 보통 숫자인 ... 를 입력했군요" 를 출력하게 되는 것이지요. 하지만, `num` 의 값이 7 이 였다면 `if` 문을 만족하는 것이기 때문에 `else` 는 거들떠 보지도 않고 넘어가게 됩니다.

```
/* if - else 죽음의 숫자? */
#include <stdio.h>
int main() {
  int num;

  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("행운의 숫자 7 이군요!\n");
  } else {
    if (num == 4) {
```

```
      printf("죽음의 숫자 4 인가요 ;;; \n");
    } else {
      printf("그냥 평범한 숫자 %d \n", num);
    }
  }
  return 0;
}
```

이번에 성공적으로 컴파일 한 후, 숫자들을 입력해 보면 비슷한 결과가 나오지만 4 를 입력했을 경우 "죽음의 숫자 4 인가요 ;;; " 가 나오게 됩니다.

실행 결과

```
아무 숫자나 입력해 보세요 : 4
죽음의 숫자 4 인가요 ;;;
```

자, 이제 소스를 뜯어 보기로 합시다.

```
else {
  if (num == 4) {
    printf("죽음의 숫자 4 인가요 ;;; \n");
  } else {
    printf("그냥 평범한 숫자 %d \n", num);
  }
}
```

앞 `if (num == 7)` 부분은 이미 설명 했으니 생략하기로 하고, `else` 문의 구조만 뜯어 보기로 합시다. 만약 `num` 의 값이 4 였다고 합시다. 그렇다면, 처음에 만나는 `if` 문에서 `num == 7` 이 거짓이 되므로 `else` 로 넘어가게 됩니다.

그런데, `else` 의 명령을 실행하려고 하는데 보니, 또 `if` 문이 있네요. 이번에는 `if (num == 4)` 가 나타납니다. 하지만 아까와는 달리 `num == 4` 가 참이므로 그 `if` 문의 중괄호 속의 명령, 즉 "죽음의 숫자 4 인가요 ;;;" 가 출력되게 됩니다.

그리고, 앞 예제에서 설명했듯이 `if(num == 4)` 아래의 `else` 는 무시 하게 되고, 끝나게 되는 것이지요.

그렇다면 이제 아이디어를 확장해서 `num` 이 1 부터 10 일 때 까지 특별한 설명을 달기로 합시다. 그러면 아래와 같이 프로그램을 짜야 되겠죠.

```
/* 쓰레기 코드 */
#include <stdio.h>
int main() {
  int num;
```

```
  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("행운의 숫자 7 이군요!\n");
  } else {
    if (num == 4) {
      printf("죽음의 숫자 4 인가요 ;;; \n");
    } else {
      if (num == 1) {
        printf("첫 번째 숫자!! \n");
      } else {
        if (num == 2) {
          printf("이 숫자는 바로 두번째 숫자 \n");
        } else {
          ......(생략).....
        }
      }
    }
  }
  return 0;
}
```

정말 믿도 끝도 없이 길어져서 나중에는 중괄호가 너무 많아 헷갈리기도 하고, 보기도 불편하게 됩니다. 하지만, C 언어는 위대한지라, 이 문제를 간단히 해결하였습니다.

```
/* 새로쓰는 죽음의 숫자 예제 */
#include <stdio.h>
int main() {
  int num;

  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("행운의 숫자 7 이군요!\n");
  } else if (num == 4) {
    printf("죽음의 숫자 4 인가요 ;;; \n");
  } else {
    printf("그냥 평범한 숫자 %d \n", num);
  }
  return 0;
}
```

위 코드를 실행해 보면 앞선 예제와 똑같이 작동합니다. 하지만 코드도 훨씬 보기 편해 졌고 난잡하던 중괄호도 어느 정도 정리가 된 것 같군요. 그렇다면 정말 하는 일이 똑같을 까요? 네, 똑같습니다. 왜냐하면

```
if (/* 조건 1 */) {
  // 명령 1;
} else {
  if (/* 조건 2 */) {
    // 명령 2;
  } else {
    if (/* 조건 3 */) {
      // 명령 3;
    } else {
      // ....
    }
  }
}
```

위와 같은 코드를 단지 아래 처럼 '간단히' 표현한 것이기 때문이죠.

```
if (/* 조건 1 */) {
  // 명령 1;
} else if (/* 조건 2 */) {
  // 명령 2;
} else if (/* 조건 3 */) {
  //명령 3;
}
....else {
  // 명령 ;
}
```

단지, 보기 편하게 하기 위해 '간략하게' 줄인 꼴과 같다는 것입니다. 마치 `a = a + 10` 을 `a+= 10` 으로 바꾼 것 처럼 말이죠.

```
/* if 와 if- else if 의 차이*/
#include <stdio.h>
int main() {
  int num;

  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("a 행운의 숫자 7 이군요!\n");
  } else if (num == 7) {
    printf("b 행운의 숫자 7 이군요! \n");
  }

  // 비교
  if (num == 7) {
    printf("c 행운의 숫자 7 이군요!\n");
```

```
  }
  if (num == 7) {
    printf("d 행운의 숫자 7 이군요! \n");
  }

  return 0;
}
```

성공적으로 컴파일 후, 7 을 입력하였다면

실행 결과

```
아무 숫자나 입력해 보세요 : 7
a 행운의 숫자 7 이군요!
c 행운의 숫자 7 이군요!
d 행운의 숫자 7 이군요!
```

와 같이 나오게 됩니다. 여기서 주목해야 할 점은, 출력되는 문장 앞의 알파벳 (a,c,d) 인데 이는 각 문장이 위 프로그램의 어느 부분에서 출력되는 지 알려줍니다.

우리가 컴퓨터 라고 생각하고 프로그램을 실행해 봅시다. 통상적으로 프로그램은 소스코드의 위에서 부터 아래 방향으로 실행됩니다

```
if (num == 7) {
  printf("a 행운의 숫자 7 이군요!\n");
} else if (num == 7) {
  printf("b 행운의 숫자 7 이군요! \n");
}
```

컴퓨터가 쭉 프로그램을 읽다가 위 부분에 도달하면

"어! `if` 문이군. `num` 의 값이 7 인지 확인해 볼까?" 라고 확인을 합니다. 그런데, 참 이므로"오, `if` 문이 참 이군. 그렇다면 중괄호 속의 내용을 실행해야겠다! " 하며 "a 행운의 숫자 7 이군요" 를 출력합니다.

그런데, 그 다음 부분인 `else if(num == 7)` 에서도 마찬가지로 num 의 값이 7 이므로 `num == 7` 이 참이 되어서 `else if` 가 참이 되어 "b 행운의 숫자 7 이군요! " 도 출력되어야 할 것 같습니다. 하지만, 결과를 보아하니 출력이 되지 않았습니다. 왜 일까요?

왜냐하면, 앞에서 누누히 설명했듯이 `else` 문은 전제 조건이 '앞의 `if` 문이 참이 아닐 때' 실행된다는 사실을 기본으로 깔고 있기 때문이죠. 따라서, 앞의 `if` 문이 참이므로 실행이 될 수 없습니다. 따라서, "b 행운의 숫자 7 이군요! " 도 출력되지 않습니다. 참고적으로 `else` 문은 언제나 `if` 문의 결과에 따라 실행 여부가 결정되므로 언제나 `else` 를 사용하려면 `if` 도 반드시 함께 따라 사용해야 합니다.

다시 컴퓨터가 쭉 읽다가 아래와 같은 문장을 발견했네요.

```
if (num == 7) {
  printf("c 행운의 숫자 7 이군요!\n");
}
if (num == 7) {
  printf("d 행운의 숫자 7 이군요! \n");
}
```

"어! if 문이군. 그런데 num 의 값이 7 이므로 이 if 문은 참이야. 중괄호 속의 내용을 실행해야지" 하면서 "c 행운의 숫자 7 이군요!" 가 출력된다. 마찬가지로 아래도

"어! if 문이군. 그런데 num 의 값이 7 이므로 이 if 문은 참이야. 중괄호 속의 내용을 실행해야지" 하면서 "d 행운의 숫자 7 이군요!" 가 출력되는 것입니다.

```
#include <stdio.h>
int main() {
  float ave_score;
  float math, english, science, programming;

  printf("수학, 영어, 과학, 컴퓨터 프로그래밍 점수를 각각 입력해 주세요 ! : ");
  scanf("%f %f %f %f", &math, &english, &science, &programming);

  ave_score =
      (math + english + science + programming) / 4;  // 4 과목의 평균을 구한다.
  printf("당신의 평균 점수는 %f 입니다 \n", ave_score);
  if (ave_score >= 90) {
    printf("당신은 우등생 입니다. ");
  } else if (ave_score >= 30) {
    printf("조금만 노력하세요!. \n");
  } else {
    printf("공부를 발로 합니까? \n");
  }

  return 0;
}
 ~/Te
```

만약 성공적으로 컴파일 하였다면

실행 결과

```
수학, 영어, 과학, 컴퓨터 프로그래밍 점수를 각각 입력해 주세요 ! : 100
90
90
85
```

```
당신의 평균 점수는 91.250000 입니다
당신은 우등생 입니다.
```

와 같이 나오게 됩니다. 그 외에도, 다른 값들을 입력하면 다른 결과가 출력됨을 알 수 있습니다.

```
ave_score = (math + english + science + programming) / 4;
```

아마, 산술 연산을 까먹으신 분들은 없겠죠? 위 식이 2 초 내로 이해가 되지 않는다면 4 강을 다시 공부하시기 바랍니다. 위 식은, 수학, 영어, 과학, 프로그래밍 점수의 평균을 구해서 ave_score 라는 변수에 대입하는 식입니다.

```
if (ave_score >= 90) {
  printf("당신은 우등생 입니다. ");
} else if (ave_score >= 40) {
  printf("조금만 노력하세요!. \n");
} else if (ave_score > 0) {
  printf("공부를 발로 합니까? \n");
} else {
  printf("인생을 포기하셨군요. \n");
}
```

위 프로그램의 핵심 부분(?) 이라 할 수 있는 이 부분을 잘 살펴봅시다. 만약 내 평균 점수가 93 이라면, "당신은 우등생 입니다" 가 출력되게 되죠. 그리고, 아래 else if 와 else 는 무시하고 종료됩니다.

하지만 평균 점수가 40 이라면, 위의 if (ave_score >= 90) 이 거짓이 되어 다음으로 넘어가죠. 즉, else if (ave_score >= 30) 을 합니다. 이번에는 참이 되므로, 조금만 노력하세요! 가 출력이 되고 종료가 됩니다.

또한, 평균점수가 10 점 이라면 위의 if 와 else if 모두 거짓이지만 else if(ave_score > 0) 은 참이 되어 공부를 발로 합니까? 가 출력 됩니다.

마지막으로 평균점수가 0 점 이하 라면, 떨거지 처리(?) 인 else 에서 참이 되어서 인생을 포기하였군요 가 실행됩니다.

```
/* 크기 비교 */
#include <stdio.h>
int main() {
  int a;
  printf("아무 숫자나 입력하세요 : ");
  scanf("%d", &a);

  if (a >= 10) {
    if (a < 20) {
      printf(" %d 는 10 이상, 20 미만인 수 입니다. \n", a);
```

```
    }
  }
  return 0;
}
```

성공적으로 컴파일 후, 10 이상 20 미만의 수를 입력했다면

실행 결과

```
아무 숫자나 입력하세요 : 10
 10 는 10 이상, 20 미만인 수 입니다.
```

와 같이 나오게 됩니다. 위 소스 코드는 간단합니다. 우리가 여태까지 배운 내용 만으로도 충분히 이해 할 수 있습니다!

```
if (a >= 10) {
  if (a < 20) {
    printf(" %d 는 10 이상, 20 미만인 수 입니다. \n", a);
  }
}
```

처음에, a 의 값이 10 이상이면 참이 되어서 중괄호 속의 내용을 실행하게 되고, 또한 거기서 a < 20 이라는 if 문을 만나게 되는데 이것 조차 참이라면 printf 가 실행되겠지요,.

그런데, 사실 위 문장은 아래와 같이 간단히 줄여 쓸 수 있습니다.

논리 연산자

```
/* 논리 연산자 */
#include <stdio.h>
int main() {
  int a;
  printf("아무 숫자나 입력하세요 : ");
  scanf("%d", &a);

  if (a >= 10 && a < 20) {
    printf(" %d 는 10 이상, 20 미만인 수 입니다. \n", a);
  }

  return 0;
}
```

위 소스를 그대로 컴파일 해 보면 위와 결과가 똑같이 나옵니다. 그렇다면 '&&' 는 무엇일까요? 그것은 바로 **논리 곱(AND)** 라고 불리는 논리 연산자 입니다.

앞에서 우리는 AND 연산에 대해 배운 적이 있었습니다. (기억이 나지 않는다면 여기를 클릭). 이 때, AND 연산의 특징은 바로 오직 `1 AND 1` 만이 결과가 1 이였고, `1 AND 0` 또는 `0 AND 0` 은 모두 결과가 0 이였습니다.

여기서도 마찬가지 입니다. 위에서도 이야기 했지만 '참' 은 숫자 1 에 대응되고 '거짓' 은 숫자 0 에 대응 됩니다. 따라서, `a >= 10` 이 참이라면 '1' 을 나타내고, 거짓이라면 '0' 을 나타냅니다.

마찬가지로, `a < 20` 도 참 이라면 '1' 을 나타내고, 거짓이라면 '0' 을 나타냅니다. 만약 `a >= 10` 도 참이고 `a < 20` 도 참 이라면 `1 AND 1` 을 연산하는 것과 같게 되어서 결과가 1, 즉 참이 되어 `if` 문의 중괄호 속의 내용을 실행하게 되죠.

반면에 `a < 10` 라던지 `a >= 20` 이여서 둘 중 하나라도 조건을 만족하지 않게 된다면 `1 AND 0` 이나 `0 AND 1` 을 하는 것과 같게 되어 결과가 `0`, 즉 거짓이 됩니다. 따라서 중괄호 속의 내용은 실행되지 않게 됩니다.

정리하자면, && 는 두 개의 조건식이 모두 '참' 이 되어야 `if` 문 속의 내용을 실행하는 것이 됩니다.

그렇다면 우리가 여태 알고 있었던 AND 연산 기호인 & 를 쓰지 않고 && 를 쓰는 것일까요? 그 이유는 & 하나는, 숫자 사이의 AND 연산을 사용 할 때 쓰고 논리 곱 연산자인 && 는 두 개 이상의 조건식 사이에서 사용됩니다.

따라서 `if (a >= 10 & a < 20 )` 이나, `0x10 && 0xAE` 와 같은 연산은 모두 틀린 것이 됩니다.

```c
/* 논리 합 */
#include <stdio.h>
int main() {
  int height, weight;
  printf("당신의 키와 몸무게를 각각 입력해 주세요 : ");
  scanf("%d %d", &height, &weight);

  if (height >= 190 || weight >= 100) {
    printf("당신은 '거구' 입니다. \n");
  }

  return 0;
}
```

성공적으로 실행 후, 키를 190 이상으로 입력했거나 몸무게를 100 이상으로 입력했다면

실행 결과

```
당신의 키와 몸무게를 각각 입력해 주세요 : 200 90
```

```
당신은 '거구' 입니다.
```

위와 같이 나오게 되죠.

이번에는 || 라는 것이 등장하였습니다. 앞서 AND 가 && 였다는 것을 보아, || 는 OR 를 나타내는 논리 연산자임을 알 수 있습니다.

기억을 되살려 봅시다. AND 와 OR 의 차이가 뭐였죠? 음.... 아, 기억 나는군요. AND 가 두 조건이 모두 참 일 때, 참을 반환한다면, OR 은 두 조건이 모두 거짓 일 때 만 거짓을 반환합니다. 다시말해 (참) || (거짓) == (참) 이 된다는 것이지요.

```
if (height >= 190 || weight >= 100)
```

그렇다면 위 if 문을 살펴 봅시다. height >= 190 이 참 이라면, OR 연산한 값은 weight 의 크기에 관계없이 무조건 참이 되어서 중괄호 속의 내용을 실행 합니다. 또한 height >= 190 이 거짓이여도, weight >= 100 이 참 이라면 중괄호 속의 내용을 실행하게 되죠.

따라서, OR 논리 연산자는 조건식에서 적어도 어느 하나가 참 이라면 무조건 if 문의 내용을 실행해 주게 됩니다.

```c
/* 논리 부정 */
#include <stdio.h>
int main() {
  int height, weight;
  printf("당신의 키와 몸무게를 각각 입력해 주세요 : ");
  scanf("%d %d", &height, &weight);

  if (height >= 190 || weight >= 100) {
    printf("당신은 '거구' 입니다. \n");
  }
  if (!(height >= 190 || weight >= 100)) {
    printf("당신은 거구가 아닙니다. \n");
  }

  return 0;
}
```

위 소스를 성공적으로 컴파일 한 후, 180 과 80 을 입력하였다면

실행 결과

```
당신의 키와 몸무게를 각각 입력해 주세요 : 180 80
당신은 거구가 아닙니다.
```

와 같이 나옵니다.

위 소스에서 관심있게 살펴 보아야 할 부분은 바로 이 부분이죠.

```
if (!(height >= 190 || weight >= 100)) {
  printf("당신은 거구가 아닙니다. \n");
}
```

(참고로 위의 소스와 다른 점은 `height` 앞에 `!` 가 붙었다는 점 입니다.) 다른 것은 다 똑같은데, 새로 붙은 `!` 가 무슨 역할을 하는 것 같나요?

아마도 예측 하셨겠지만, `!` 는 NOT 을 취해주는 연산자 입니다. 다시 말해, **반전**을 시켜 주는 것이지요. 위 경우, `height >= 190 || weight >= 100` 의 가 **거짓**일 경우에만, 다시 말해서 `height < 190 && weight < 100` 인 경우에만 중괄호 속의 내용이 실행 됩니다. 어때요, 간단하죠?

이것으로 C 언어에서 중요한 부분인 `if` 문에 대해 알아 보았습니다. 이제 마지막으로 할 일은 바로 `if` 문을 가지고 여러가지 프로그램을 만들어 보는 것입니다. 솔직히 말해서, `if` 문 이전까지 배운 내용으로는 그다지 할 것들이 없었습니다.

하지만, 이번 `if` 문을 통해서 컴퓨터에게 '생각(?)' 할 수 있는 능력을 부여 할 수 있게 됩니다. 따라서, 무궁 무진한 것들을 만들어 낼 수 있게 되죠. 뭐, 만들어 볼 것들로 제가 추천하는 것들은 계산기나 성적표라던지, 등등. 아무튼, 힘내세요.

뭘 배웠지?

- `if, else if, else` 가 무엇 인지 알고 있습니다.
- 논리 연산자 `&&, ||` 를 배웠습니다.
- `!` 의 역할을 알고 있습니다.
- `0 <= a <= 1` 을 잘못된 사용 예 입니다. 이 대신 `0 <= a && a <= 1` 과 같이 사용해야 합니다.

6

반복문 (for, while)

안녕하세요, 여러분. C 언어 공부는 잘 되가고 있나요? 제 사이트의 강좌 만을 보고도 충분히 C 언에 대한 깊은 학습이 이루어 질 수 있다고 생각하지만 사이트의 강좌 업데이트 속도가 느리니 답답하신 분들도 많을 것 입니다. 그러한 분들은 특별히 C 언어 관련 책을 서점에서 구매하여 읽어보는 것을 추천합니다.

물론 학원을 다녀도 되지만, 시중에 C 언어에 관련한 훌륭한 학습서 (흔히 가장 많이 추천하는 것으로는 열혈강의 C 프로그래밍, A Book On C, Teach Yourself C, 등등) 들이 많이 있으므로 굳이 학원을 다닐 필요가 없을 것이라 생각합니다. 또한 인터넷을 통해서도 C 언어에 관해 많은 정보를 알 수 있으니 이점 유의하시기 바랍니다.

3일전에, Psi 는 친구로 부터 1 부터 100 까지의 합을 구해달라는 요청을 받았습니다. Psi 는 수학자 가우스 처럼 똑똑하지가 못하기에, 등차수열의 합을 구하는 방법을 알지 못했습니다. 하지만 Psi 는 이 블로그에서 C 언어를 통해 계산하는 법을 알았으므로 이를 이용하기로 하였습니다. 그래서, 그는 다음과 같이 30분 동안 열심히 타이핑 하여 아래와 같은 프로그램을 만들었습니다.

```c
#include <stdio.h>
int main() {
  printf("%d", 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 +
                16 + 17 + 18 + 19 + 20 + 21 + 22 + 23 + 24 + 25 + 26 + 27 +
                28 + 29 + 30 + 31 + 32 + 33 + 34 + 35 + 36 + 37 + 38 + 39 +
                40 + 41 + 42 + 43 + 44 + 45 + 46 + 47 + 48 + 49 + 50 + 51 +
                52 + 53 + 54 + 55 + 56 + 57 + 58 + 59 + 60 + 61 + 62 + 63 +
                64 + 65 + 66 + 67 + 68 + 69 + 70 + 71 + 72 + 73 + 74 + 75 +
                76 + 77 + 78 + 79 + 80 + 81 + 82 + 83 + 84 + 85 + 86 + 87 +
                88 + 89 + 90 + 91 + 92 + 93 + 94 + 95 + 96 + 97 + 98 + 99 +
                100);
  return 0;
}
```

그리고, 무사히 계산 결과인 5050을 구해서 친구에게 알려주었습니다. 그런데, 이게 웬일입니까? 그 친구가 갑자기 1 부터 10000까지의 숫자의 합을 계산해 달라고 요청하는 것 아니겠습니까? 위 1

부터 100 까지 쓰는 것도 힘들어 죽겠는데 10000 까지 라니. Psi 는 눈 앞이 캄캄하였습니다. 적어도, 이 강좌를 보기 전 까지는 말이죠.

for 문 (for statement)

여러분은 컴퓨터가 왜 생겨났는지 아십니까? 물론 여러가지 이유가 있겠지만 그 중 제일 중요한 이유는 바로 계산 입니다.최초의 컴퓨터라고 알려진 ENIAC (물론 이에 대해 의견이 분분 하지만 가장 일반적으로 최초의 컴퓨터는 ENIAC 이나 영국의콜로서스 둘 중 하나이네요) 은 탄도의 발사표를 계산하는 역할을 하였습니다. 그렇다면 두 번째로 중요한 컴퓨터의 존재 이유는 무엇일까요? 바로, 노가다 - 즉 반복 연산 입니다.

예를 들어서, 1 부터 100 까지 곱한다고 칩시다. 인간은 지능이 있으므로 충분한 시간만 주어진다면 이를 수행할 수 있습니다. 단, 엄청난 짜증을 내겠지요. 그리고 '도대체 이런 계산을 내가 왜 하나?' 라는 생각도 들어 계산을 하다 말고도망갈 수 도 있습니다. 하지만 컴퓨터의 경우 그렇지 않습니다. 우리가 어떤 생-노가다 성 일을 시켜도 묵묵히 자기 일만합니다. 아무리 지겨운 연산 이라도 전기 조금 더 달라는 요구, 조금 쉬게 해달라는 요구도 없이 묵묵히 자기 일 만 할뿐이지요. 이 것이 바로 우리가 컴퓨터를 쓰는 두 번째 이유 입니다.

따라서, 이번 강좌에서는 반복문에 대해 중점적으로 알아보도록 하겠습니다. 반복문은 컴퓨터 상에서 상당히 많이 쓰이므로 반드시 이해하시기 바랍니다. 일단, C 언어에서 사용할 수 있는 반복문은 여러 종류가 먼저 있습니다만, 가장 먼저 널리 쓰이는 for 문에 대해 알아 보도록 하겠습니다.

```c
#include <stdio.h>
int main() {
  int i;
  for (i = 0; i < 20; i++) {
    printf("숫자 : %d \n", i);
  }

  return 0;
}
```

위 소스를 성공적으로 컴파일 하였다면,

실행 결과

```
숫자 : 0
숫자 : 1
숫자 : 2
숫자 : 3
숫자 : 4
```

```
숫자 : 5
숫자 : 6
숫자 : 7
숫자 : 8
숫자 : 9
숫자 : 10
숫자 : 11
숫자 : 12
숫자 : 13
숫자 : 14
숫자 : 15
숫자 : 16
숫자 : 17
숫자 : 18
숫자 : 19
```

와 같이 나옵니다.for 문은 다음과 같은 기본 구조를 가지고 있습니다.

```
for (/* 초기식 */; /* 조건식 */; /* 증감식 */) {
  // 명령1;
  // 명령2;
  // ....
}
```

일단, 각 부분의 역할이 무엇인지 알아 보도록 하죠. 초기식에서는 제어변수가 초기화 됩니다. 이 말은 즉슨, for 문은 반복문이고, 반복문은 얼마나 반복을 해야 될 지 알아야 합니다. 만약 반복문이 끊이지 않고 반복한다면 CPU 사용률을 100%로 끌어 올려 전력 낭비일 뿐 만이 아니라 코드 뒷부분이 실행되지 않아 여러 오류들이 발생될 수 있습니다.

따라서, C 언어에서는 반복문이 얼마나 반복해야 할 지를 알기 위해 '제어변수' 라는 것을 도입하였습니다. for 문으로 하여금 제어변수가 특정한 조건을 만족할 때 에만 반복을 계속하게 한다는 것입니다. 제어변수의 초기값은 for 문의 '초기식' 부분에서 지정 됩니다. 예를 들어서 내가 i 를 제어변수로 이용하다면 초기식에 i = 4; 가 되면 처음 i 의 값을 4 로 한다는 뜻이지요.

이제, for 문이 조건식을 봅니다. 조건식은 우리의 제어 변수인 i 가 만족해야 될 특정한 조건이 있습니다. 예를들어, i 의 값은 언제나 10 미만이라 던지 (i < 10), i 는 언제나 1 이상 이라던지 (i >= 1). for 문은 이러한 조건식이 참 일때에만 그 일을 수행합니다. 여기서 '그 일' 은 중괄호 속의 명령들을 실행한다는 뜻이지요.

마지막으로 증감식은 "1 회 실행 시 i 의 값을 어떻게 만들어야 되냐? " 가 나타나 있습니다. 예를 들어서 증감식에 i ++ 이 써 있다면 한 번 실행 할 때 마다, i 의 값을 1 증가 시킵니다.

마찬가지로 증감식 부분에 `i -= 2` 라면 한 번 실행 할 때 마다 2 씩 감소하겠네요.

매번 실행 할 때 마다, `for` 문의 증감식이 실행 되고, 그다음에 조건식을 체크 합니다. 만약에 조건식이 `i < 10` 이였고, `i` 의 값은 9 였다고 칩시다. 또한 증감식이 `i ++` 이였다면, 명령들을 실행 한 후, 증감식이 실행되어 `i` 의 값은 10 이 됩니다. 따라서, 조건식이 거짓이 되어 `for` 문을 빠져 나갑니다.

그렇다면 위의 소스 코드는 어떨까요?

```
for (i = 0; i < 20; i++) {
  printf("숫자 : %d \n", i);
}
```

우리가 컴퓨터라면, 일단 컴퓨터는 `for` 문을 보고,

> 음, `i` 의 값을 0 으로 해야 겠다. (초기식)

`for` 문에 `i < 20` 으로 되어 있으므로 (조건식)

> `i < 20` 이 맞나? 맞네.. 그럼 중괄호 속의 내용을 실행해야지. 숫자 0 출력!

또한, `for` 문에 `i++` 로 되어 있으므로 (증감식)

> 이제 `i` 의 값을 1 증가 시켜야 겠다.

따라서, `i` 의 값은 1 이 된다.

> i 의 값이 20 미만 인가? 어, 맞네. 그러면 한 번 더 실행 해야 겠다. (조건식) 숫자 1 출력

..... (생략) .....

> 이제 `i` 의 값을 1 증가 시켜야 겠다 (증감식)

20 번의 실행 후, `i` 의 값이 마침내 20 이 되었다.

> i 의 값이 20 미만 인가? 어? 아니잖아. (조건식) 그러면 이제 `for` 문을 빠져 나가야지

하며, 더이상 중괄호 속의 내용을 실행하지 않는다. 숫자 *20* 이 출력 되지 않는다.

`for` 문은 의외로 간단 합니다. 단지 기억 하실 것은 **`for` 문은 `{}` 안에 작업들을 조건식이 성립할 동안 반복해주는 것이고, 매 반복마다 증감식을 실행한다** 라고 이해 하시면 되겠습니다.

```c
/* 1 부터 19 까지의 합*/
#include <stdio.h>
int main() {
  int i, sum = 0;
  for (i = 0; i < 20; ++i) {
    sum = sum + i;
  }
  printf("1 부터 19 까지의 합 : %d", sum);

  return 0;
}
```

위 소스를 성공적으로 컴파일 했다면

실행 결과

```
1 부터 19 까지의 합 : 190
```

와 같이 나옵니다.

만약 위 결과를 믿지 못하는 사람들은 직접 계산기로 더하거나, 등차수열의 합 공식을 이용하여 직접 셈하셔 보시기 바랍니다. 아마, 독자 여러분들의 컴퓨터가 비정상이 아니라면, 아니면 당신의 눈이 잘못되지 않는 한 위 결과는 190 으로 나올 것 입니다.

일단, 위 프로그램의 핵심부분은 아래와 같습니다.

```c
for (i = 0; i < 20; ++i) {
  sum = sum + i;
}
```

for 문을 살펴보자면, 위 for 문은 총 20 회 실행되며 i 는 0 부터 19 까지의 값을 가집니다. 이 때 주목해야 할 부분은 바로

```c
sum = sum + i;
```

이 부분이죠. sum 이라는 변수에 i 의 값이 계속 더해집니다. 아시다 싶이 여러분은 sum = sum + i 라는 식의 뜻이 0 = i 라는 괴상한 방정식이 아니라 '=' 를 '대입 연산자' 로 생각하여 'sum 이란 변수에 sum + i 의 값을 집어 넣는다' 라는 의미가 됩니다. 즉, 위 상태로 for 문을 실행하게 되면 sum 에 0 부터 19 까지의 값이 더해지게 됩니다.

위 for 문을 보통 수식으로 풀어쓰면 아래와 같이 됩니다.

```
sum = 0;  // 초기 조건
sum = sum + 0;
sum = sum + 1;  // sum = 1;
sum = sum + 2;  // sum = 3;
sum = sum + 3;  // sum = 6;
// ....
sum = sum + 19;  // sum = 190;
```

이 되는 것이지요. 그렇다면 이제 Psi 의 고충을 풀어줄 시간이 왔네요. 1 부터 10000 까지의 합은 어떻게 구할까요? 그야 간단합니다. 단지 조건식만 약간 수정해 주면 됩니다. 한가지 걱정할 부분은 만약 10000 까지의 합이 `int` 자료형의 범위보다 크면 안되는데, 다행히도 크지 않으므로 그냥 계산하시면 됩니다. 이는 아래와 같습니다.

```c
#include <stdio.h>
int main() {
  int i, sum = 0;
  for (i = 0; i <= 10000; ++i) {
    sum = sum + i;
  }
  printf("1 부터 10000 까지의 합 : %d \n", sum);

  return 0;
}
```

그 결과는

실행 결과

```
1 부터 10000 까지의 합 : 50005000
```

와 같네요. 결국 Psi 는 친구와의 우정을 지킬 수 있었습니다. ㅎㅎ

```c
/* for 문 응용 */
#include <stdio.h>
int main() {
  int i;
  int subject, score;
  double sum_score = 0;

  printf("몇 개의 과목 점수를 입력 받을 것인가요?");
  scanf("%d", &subject);

  printf("\n 각 과목의 점수를 입력해 주세요 \n");
  for (i = 1; i <= subject; i++) {
```

```
    printf("과목 %d : ", i);
    scanf("%d", &score);
    sum_score = sum_score + score;
  }

  printf("전체 과목의 평균은 : %.2f \n", sum_score / subject);

  return 0;
}
```

위 소스를 성공적으로 컴파일 하였다면

실행 결과

```
몇 개의 과목 점수를 입력 받을 것인가요?4


 각 과목의 점수를 입력해 주세요
과목 1 : 100
과목 2 : 99
과목 3 : 89
과목 4 : 76
전체 과목의 평균은 : 91.00
```

음, 여러 과목을 입력해 보면서 실제 시험 성적 평균을 내보시기 바랍니다. 아무튼, 위 소스를 살펴봅시다. 일단, 가장 중요한 부분인 `for` 문 부분 부터 보자면...

```
for (i = 1; i <= subject; i++) {
  printf("과목 %d : ", i);
  scanf("%d", &score);
  sum_score = sum_score + score;
}
```

`for` 문을 살펴보면 `i` 의 값이 1 에서 `subject` 까지 1 씩 증가하면서 돌아가네요. 이 말은 즉슨, `for` 문 안의 내용이 `subject` 번 실행된다는 뜻입니다. (즉, `subject` 가 3 이라면, `i` 의 값이 1 부터 3 까지 1 씩 증가하면서 돌아가므로 `1,2,3.` 즉 3 번 `for` 문 속 내용이 실행됩니다)

이 때,

```
printf("과목 %d : ", i);
scanf("%d", &score);
```

위 부분에서 각 과목의 점수를 입력받고, 그 입력받은 점수를 `score` 라는 변수에 저장하게 되죠.

```
sum_score = sum_score + score;
```

그리고, 그 입력받은 score 를 sum_score 에 더하게 됩니다. 다시말해, for 문이 모두 돌아가고 나면 sum_score 에는 입력받은 과목들의 점수의 합이 들어가게 됩니다. 따라서,

```
printf("전체 과목의 평균은 : %.2f \n", sum_score / subject);
```

평균은 총점을 과목 수로 나눈 것이므로 sum_score / subject 가 우리가 구하고 싶은 전체 과목 평균이 되겠군요.

break 문

```
/* break! */
#include <stdio.h>
int main() {
  int usranswer;

  printf("컴퓨터가 생각한 숫자를 맞추어 보세요! \n");

  for (;;) {
    scanf("%d", &usranswer);
    if (usranswer == 3) {
      printf("맞추셨군요! \n");
      break;
    } else {
      printf("틀렸어요! \n");
    }
  }

  return 0;
}
```

성공적으로 실행했다면 아래와 같이 나오게 됩니다.

실행 결과

```
컴퓨터가 생각한 숫자를 맞추어 보세요!
5
틀렸어요!
6
틀렸어요!
```

```
7
틀렸어요!
8
틀렸어요!
3
맞추셨군요!
```

3 이 입력 될 때 까지 계속 물어보다가 3 을 입력하게 되면 프로그램이 종료됩니다 3 이 입력 될 때 까지 계속 물어보다가 3 을 입력하게 되면 프로그램이 종료됩니다.

```c
for (;;) {
  scanf("%d", &usranswer);
  if (usranswer == 3) {
    printf("맞추셨군요! \n");
    break;
  } else {
    printf("틀렸어요! \n");
  }
}
```

일단, for 문을 잠시 살펴 봅시다. 그런데, 한 가지 이상한 점이 있죠? 초기식, 조건식, 증감식이 모두 없습니다! 그렇다면 이 for 문은 얼마나 실행 되야 되는 것인가요? 답은, for 문의 조건식이 명시되지 않는다면 항상 **참** 이라 인식 되기 때문에 이 for 문은 언제나 참이됩니다. 다시 말해, 무한히 중괄호 속의 내용을 실행한다는 것이지요. 그래서, 만약

```c
#include <stdio.h>
int main() {
  for (;;) {
    printf("a");
  }
  return 0;
}
```

와 같은 프로그램을 만든다면, for 문이 무한번 실행되므로 (프로그램 자체가 강제적으로 종료되기 전 까지)

실행 결과

```
aaaa
```

```
scanf("%d", &usranswer);
if (usranswer == 3) {
  printf("맞추셨군요! \n");
  break;
} else {
  printf("틀렸어요! \n");
}
```

아무튼, scanf 를 통해 usranswer 에 사용자가 입력한 수를 저장합니다. 그리고 if 문을 통해 비교하지요. 과연 컴퓨터가 생각한 3 과 같은지... 만약 같다면 '맞추셨군요!' 가 출력이 됩니다. 그리고,프로그램이 종료되죠. 즉, for 문을 빠져나갑니다. 그런데, 맞추지 못하면 for 문은 계속 돌고 돌게 됩니다. 우리가 맞출 때 까지요. 그렇다면, 위 소스에서 for 문을 빠져나가게 하는 부분은 무엇일까요. 아마 짐작했던 대로, break 가 for 문을 탈출 시킵니다.

break 는 for 문에 조건식에 상관 없이 실행이 되기만 하면 for 문을 그대로 탈출 시켜 버립니다. 이 말은 즉슨 break 아래의 어떠한 것들도 실행이 되지 않는 다는 것이지요.

```
#include <stdio.h>
int main() {
  for (;;) {
    break;
    printf("a");
  }
  return 0;
}
```

따라서, 위와 같은 프로그램을 만들었을 때, break 문을 만나자 마자 for 문 밖으로 탈출 시키므로 a 는 출력이 되지 않고 프로그램은 종료됩니다. 반면에

```
#include <stdio.h>
int main() {
  for (;;) {
    printf("a");
    break;
  }
  return 0;
}
```

위와 같이 break 앞에 printf("a"); 가 있다면 a 가 출력이 되고 for 문을 빠져나가 종료가 되는 것이지요.

사실, 무한 for 문은 생소하기는 해도 많은 곳에서 쓰이고 있습니다. 예를 들어 어떤 게임에서

```
for (;;) {
  // 게임;
  if (/* 유저 사망 */) {
    if (/* 게임 다시 안할래요 */) {
      break;
    }
  }
  // 게임 재시작;
}
```

와 같이 쓰일 수 있습니다.

continue 문

continue 문은 break 문과 비슷하지만서도 하는 일은 완전히 다릅니다. continue 는 break 와는 달리 for 문을 빠져 나가지 않고, 그냥 패스 해주는 것입니다. 아래 예제를 봅시다.

```
/* 5 의 배수를 제외한 숫자 출력 */
#include <stdio.h>
int main() {
  int i;

  for (i = 0; i < 100; i++) {
    if (i % 5 == 0) continue;

    printf("%d ", i);
  }

  return 0;
}
```

성공적으로 실행하면

실행 결과

```
1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19 21 22 23 24 26 27 28 29 31
↪ 32 33 34 36 37 38 39 41 42 43 44 46 47 48 49 51 52 53 54 56 57
↪ 58 59 61 62 63 64 66 67 68 69 71 72 73 74 76 77 78 79 81 82 83
↪ 84 86 87 88 89 91 92 93 94 96 97 98 99
```

와 같이 나오게 됩니다. 보시다 싶이, 5 의 배수를 제외한 0 이상, 100 미만의 모든 수들이 출력되었습니다.

```c
for (i = 0; i < 100; i++) {
  if (i % 5 == 0) continue;

  printf("%d ", i);
}
```

일단, for 문을 살펴보면 i 가 0 부터 100 미만의 값을 가지게 됩니다. 이 때, if 문을 살펴 보면 i 를 5 로 나눈 나머지 (i % 5) 가 0 일 때 (== 0), continue 를 실행함을 볼 수 있습니다.

continue 는 break 문 처럼 아래 모든 내용을 무시한다는 점에서 동일하지만, break 문은 루프를 빠져나가는데 반면 continue 는 다시 조건 점검부로 점프하게 됩니다. continue 는 마치 카드게임에서 스킵과 같은 역할을 하게 됩니다. (break 문이 카드게임에서 퇴출 되는 것이라면...)

따라서, i 의 값이 5 의 배수인 경우에만 printf("%d", i) 가 실행이 되지 않게 되는 것이지요.

문득 for 문을 배우면서 이러한 생각은 들지 않았나요? if 문 안에 if 문을 넣을 수 있는 것 처럼 for 문 안에도 for 문을 넣을 수 있을까? 네, 물론 넣을 수 있습니다. 아래 예제를 참조하세요.

```c
/* 구구단 */
#include <stdio.h>
int main() {
  int i, j;

  for (i = 1; i < 10; i++) {
    for (j = 1; j < 10; j++) {
      printf(" %d x %d = %d \n", i, j, i * j);
    }
  }

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
.... (생략) ...
 7 x 8 = 56
 7 x 9 = 63
 8 x 1 = 8
 8 x 2 = 16
 8 x 3 = 24
 8 x 4 = 32
```

```
  8 x 5 = 40
  8 x 6 = 48
  8 x 7 = 56
  8 x 8 = 64
  8 x 9 = 72
  9 x 1 = 9
  9 x 2 = 18
  9 x 3 = 27
  9 x 4 = 36
  9 x 5 = 45
  9 x 6 = 54
  9 x 7 = 63
  9 x 8 = 72
  9 x 9 = 81
```

와 같이 근사한 구구단 표가 출력됩니다.

```
for (i = 1; i < 10; i++) {
  for (j = 1; j < 10; j++) {
    printf(" %d x %d = %d \n", i, j, i * j);
  }
}
```

위 코드에서 구구단 표를 출력하는 부분은 바로 위 부분입니다. `for` 문이 2 개나 사용되어 있는 꼴이지요. 그런데 사실 돌아가는 원리는 간단합니다. 일단, 처음에 `i` 에 1 이 들어 가게 되죠. 그런 다음에

```
for (j = 1; j < 10; j++) {
  printf(" %d x %d = %d \n", i, j, i * j);
}
```

이 부분이 열심히 실행됩니다. 물론 위 부분이 열심히 실행되는 동안 `i` 의 값은 변하지 않고 (계속 1 로 남는다), `j` 의 값만 1 부터 9 까지 변하여 구구단의 `1 x 1` ~ `1 x 9` 까지 출력하게 되는 것이지요. 위 `for` 문이 끝나면, 다시

```
for (i = 1; i < 10; i++)
```

이 부분이 실행되어 `i` 의 값이 1 증가합니다. 즉, `i` 는 2가 되는 것이지요. 이제 다시

```
for (j = 1; j < 10; j++) {
  printf(" %d x %d = %d \n", i, j, i * j);
}
```

가 실행되어 2 x 1 ~ 2 x 9 까지 출력되게 됩니다. 마찬가지 방법으로 i 의 값이 9 가 될 때까지 실행한 뒤 i 의 값이 10 이 되면 for 문을 완전히 빠져 나와 실행이 종료 됩니다.

```
/* 다음 소스만 보고 무슨 숫자가 출력될 지 맞추어 보세요~~ */
#include <stdio.h>
int main() {
  int i, j;

  for (i = 1; i < 10; i++) {
    for (j = 1; j < i; j++) {
      printf("%d ", j);
    }
  }

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 1 2 3 4 5 6 1 2 3 4 5 6 7 1 2 3 4 5
↪  6 7 8 %
```

가 나오게 됩니다. 아마 위에서 for 문에 대해 잘 이해하신 분들은 금방 이해 할 수 있겠지요.

```
for (i = 1; i < 10; i++) {
  for (j = 1; j < i; j++) {
    printf("%d ", j);
  }
}
```

이 부분에서 i 가 1 이면, j 가 출력되지 않고, i 가 2 가 되면 j 가 1 부터 1 까지, i 가 3 이 되면 j 는 1 부터 2 까지 순차적으로 출력되어 i 가 9 일 때, j 는 1 부터 8 까지 출력되어 위와 같은 모습을 보이게 됩니다. 어때요? 간단하지요?

while 문

아마 이 쯤 하셨다면 for 문에 대해 질렸을 것 같으니 for 문과 비스므리하면서도 다른 반복문인 while 문에 대해 살펴 보도록 해봅시다.

```c
/* while 문 */
#include <stdio.h>
int main() {
  int i = 1, sum = 0;

  while (i <= 100) {
    sum += i;
    i++;
  }

  printf("1 부터 100 까지의 합 : %d \n", sum);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
1 부터 100 까지의 합 : 5050
```

와 같이 1 부터 100 까지 숫자들의 합이 출력됩니다.

while 문은 위의 예제에서도 알 수 있듯이 for 문과는 달리 구조가 사뭇 단순합니다. while 문의 기본 구조는 아래와 같습니다.

```c
while (/* 조건식 */) {
  // 명령1;
  // 명령2;
  // ...
}
```

for 문 처럼 '조건식' 에는 이 while 문을 계속 돌게 할 조건이 들어갑니다. 예를 들어서 조건식에 i <= 100 이 들어간다면 i 가 100 이하 일 때 만 조건이 성립하므로 i 가 100 이하일 때 까지 while 문이 계속 돌아가게 됩니다.

```c
while (i <= 100) {
  sum += i;
```

```
  i++;
}
```

위 경우, `i` 의 값이 100 이하 인 지 검사한 다음에 (`i <= 100`), `sum` 에 `i` 를 더하고 (`sum += i`), `i` 의 값을 증가한 뒤 (`i++`), 다시 처음으로 돌아가게 됩니다. 이 때, `while` 문의 특징이 바로 시작부터 조건식을 검사한다는 것입니다. (이는 `for` 문과 동일합니다.)

따라서, 만약 `i < 1` 이 조건식이라면 `while` 문 내부의 내용은 하나도 실행되지 않고 종료되게 됩니다.

do-while 문

```c
#include <stdio.h>
int main() {
  int i = 1, sum = 0;

  do {
    sum += i;
    i++;
  } while (i < 1);

  printf(" sum : %d \n", sum);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
 sum : 1
```

와 같이 나오게 됩니다.

`do-while` 문의 구조는 아래와 같습니다.

```c
do {
  // 명령1;
  // 명령2;
  // ...
} while (/* 조건식 */);
```

`do - while` 문은 사실 `while` 문과 거의 비슷합니다. 한 가지 차이점은 앞서 말했듯이 `while` 문은 명령을 실행하기 전에 조건식이 참 인지 먼저 검사 합니다. 따라서, 조건식이 처음부터 참이 아니라면 `while` 문 안의 내용은 결코 실행 될 수 없겠지요.

그런데, `do - while` 은 먼저 명령을 실행 한 뒤에 조건식을 검사합니다. 따라서, 처음부터 조건식이 참이 아니라도 명령을 먼저 실행한 다음 조건식을 검사하기 때문에 최소한 한 번은 실행되게 됩니다.

```
do {
  sum += i;
  i++;
} while (i < 1);
```

따라서, 위 경우 `i` 가 1 로 `i < 1` 이 였지만 조건식을 나중에 검사하기 때문에 일단 `sum + = i;` 와 `i ++` 을 실행 한 다음에 `i < 1` 이 검사되어 `sum` 의 값이 1 이 출력될 수 있었던 것이지요. 어때요, 간단하죠?

그렇다면 이제 반복문에 대해 대충 감을 잡았을 것으로 기대합니다. 하지만 사실 반복문을 익숙하게 사용할 때 까지 많은 연습이 필요하기 때문에 제가 아래 반복문 사용법을 연습할 수 있는 몇 개 문제들을 준비하였습니다. 처음에는 문법 자체가 어색하므로 시간이 좀 걸리겠지만, 스스로 해보시길 바랍니다!

생각 해보기

문제 1 (난이도 : 中)

`N` 줄인 삼각형을 출력한다. 단, 사용자로 부터 임의의 `N` 을 입력 받는다. 아래는 `N = 3` 일 때의 출력 예시 이다.

```
* *** *****
```

문제 2 (난이도 : 中上)

위와 동일한 형태를 취하되, 역 삼각형을 출력한다. 아래는 `N = 3` 일 때의 출력 예시 이다.

```
***** *** *
```

문제 3 (난이도 : 下)

1000 이하의 3 또는 5 의 배수인 자연수들의 합을 구한다.

문제 4 (난이도 : 中)

1000000 이하의 피보나치 수열 (`N` 번째 항이 `N - 1` 번째 항과 `N - 2` 번째 항으로 표현되는 수열, 시작은 1,1,2,3,5,8,...) 의 짝수 항들의 합을 구한다

문제 5 (난이도 : 下)

사용자로 부터 `N` 값을 입력 받고 1 부터 `N` 까지의 곱을 출력한다.

문제 6 (난이도 : 中)

다음 식을 만족하는 자연수 `a,b,c` 의 개수를 구하여라

i) $a + b + c = 2000$ ii) $a > b > c$, a,b,c 는 모두 자연수

문제 7 (난이도 : 中上)

임의의 자연수 `N` 을 입력 받아 `N` 을 소인수 분해 한 결과를 출력하여라. 예를 들어서 `N = 18` 일 경우

```
N = 1818 = 2 * 3 * 3
```

문제 8 (난이도 : 上)

문제 7 에서 만든 프로그램의 속도를 향상 시킬 수 있는 방법은 없을까? 큰 수를 빠르게 소인수분해 할 수 있는 방법들을 찾아 프로그램에 적용시켜 보아라. 예를 들어서 `N` 의 제곱근 이하의 정수들만 처리한다던지, `Lucas- Lehmer` 판정법을 이용해 소수인지 아닌지 판정한다던지 등등..

(참조 : * 표가 붙은 문제들은 http://projecteuler.net/index.php?section=problems 에서 가져온 것들)

뭘 배웠지?

- `for` 문과 `while` 문의 사용법을 이해하고 사용할 수 있습니다.
- `break` 문을 사용하면 가장 가까운 루프에서 빠져나갈 수 있습니다.
- `continue` 를 사용하면, 아래 내용을 실행하지 않고 다음 루프를 실행합니다.

리눅스에서 C 프로그래밍 하기

7

switch 문

안녕하세요 여러분. 그 동안 잘 지내셨는지요? 제가 그 동안 바빠서 글을 많이 못 올렸으나 일이 잘 해결되어서 이제 더이상 이전처럼 바쁘지 않겠네요. 아무튼, 지금까지 제 강좌를 보시느라 오랫동안 기다려 주신 분들께 정말 감사하다고 생각되고 아직까지도 C 언어를 배우고자 하는 열정이 사그라들지 않은 여러분들은 최고의 C 언어 프로그래머가 될 것이라 믿습니다.

이번 강좌에서는 `if` 문의 친구인 `switch` 문에 대해 배워 보도록 하겠습니다. `switch` 문이 `if` 문의 친구라고 한 이유는 하는 일이 정말로 `if` 문과 비슷하기 때문이죠. 일단, 아래의 초-간단한 강아지 시뮬레이션을 보세요.

```c
/* 마이펫 */
#include <stdio.h>
int main() {
  int input;

  printf("마이펫 \n");
  printf("무엇을 하실 것인지 입력하세요 \n");
  printf("1. 밥주기 \n");
  printf("2. 씻기기 \n");
  printf("3. 재우기 \n");

  scanf("%d", &input);

  if (input == 1) {
    printf("아이 맛있어 \n");
  } else if (input == 2) {
    printf("아이 시원해 \n");
  } else if (input == 3) {
    printf("zzz \n");
  } else {
    printf("무슨 명령인지 못 알아 듣겠어. 왈왈 \n");
  }
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
마이펫
무엇을 하실 것인지 입력하세요
1. 밥주기
2. 씻기기
3. 재우기
1
아이 맛있어
```

와 같이 3 가지 명령에 대해 반응하고 알 수 없는 명령은 '무슨 명령인지 못 알아 듣겠어. 왈왈' 라고 내보냅니다.

그런데, 만약 강아지가 위 3 가지 명령만 반응하는 것이 아니라 10 가지 명령에 반응하게 하고 싶다고 합시다. 그렇다면 여러분은 아마도 아래와 같이 할 것 입니다. (참고로 아래 '...' 인 부분은 필자가 쓰기 귀찮아서 생략한 부분 입니다.)

```
if (...) {
  ...
} else if (...) {
  ...
} else if (...) {
  ...
} else if (...) {
  ...
} else if (...) {
  ...
} else if (...) {
  ...
} else if (...) {
  ...
} else if (...) {
  ...
} else if (...) {
  ...
} else if (...) {
  ...
}
```

음, 아마도 위 소스코드를 보는 사람이 상당히 불편하게 느낄 것이라고 생각되지 않나요? 물론 보는 이에 따라 다르겠지만 아마 대부분의 사람이 그렇게 생각할 것 입니다. 아마 실제로 사람들과 함께 프로젝트를 진행 할 때 위와 같은 소스를 남발하게 된다면 읽는이도 불편하고 쓰는 사람도 손목이 많이 아플 것 입니다. (물론 `Ctrl + v` 신공이 있기는 하지만...)

따라서, 위와 같이 동일한 변수에 대해 비교문이 반복되는 경우에 아래와 같이 깔끔한 switch 문을 적용 시킬 수 있습니다.

```c
/* 업그레이드 버전 */
#include <stdio.h>
int main() {
  int input;

  printf("마이펫 업그레이드\n");
  printf("무엇을 하실 것인지 입력하세요 \n");
  printf("1. 밥주기 \n");
  printf("2. 씻기기 \n");
  printf("3. 재우기 \n");

  scanf("%d", &input);

  switch (input) {
    case 1:
      printf("아이 맛있어 \n");
      break;

    case 2:
      printf("아이 시원해 \n");
      break;

    case 3:
      printf("zzz \n");
      break;

    default:
      printf("무슨 명령인지 못 알아 듣겠어. 왈왈 \n");
      break;
  }

  return 0;
}
```

아마 컴파일 된 결과는 위와 동일하게 나올 것 입니다. 이제, 위 소스 코드에서 가장 중요한 부분인 switch 문 부분을 살펴보도록 합시다.

```c
switch (input) {
  case 1:
    printf("아이 맛있어 \n");
    break;

  case 2:
    printf("아이 시원해 \n");
    break;
```

```
  case 3:
    printf("zzz \n");
    break;

  default:
    printf("무슨 명령인지 못 알아 듣겠어. 왈왈 \n");
    break;
}
```

switch 문의 기본 구조는 아래와 같습니다.

```
switch (/* 변수 */) {
  case /* 값1 */:
    // 명령들;
    break;
  case /* 값2 */:
    // 명령들;
    break;
    //.. (생략) ..
}
```

이 때, 변수 부분에는 값1, 값2, ... 들과 비교할 변수가 들어가게 됩니다. 위 예제의 경우 input 을 1 과 2 와 3 과 비교해야 했으므로 변수 부분에는 input 이 들어가게 됩니다. 이 때 switch 문에 사용될 변수로는 반드시 정수 데이터를 보관하는 변수여야 합니다. 다시말해 '변수' 부분에 들어가는 변수들의 타입은 char, short, int, long 중의 하나여야 합니다. 만약 input 이 float 이나 double 이라면 컴파일시 오류가 발생되게 됩니다.

변수 == 값1 일 때, 가장 맨 위의 case 의 명령이 실행됩니다. 위 예제의 경우 1 이 입력되면 case 1: 이 참이 되므로 그 case 안의 내용들이 모두 실행됩니다. 이 때 각 명령들을 모두 실행한 후 break 를 만나면 switch 문을 빠져 나가게 됩니다.

예를 들어서 1 이 입력되었다면 case 1: 이 참이므로 printf("아이 맛있어 \n"); 와 break; 가 실행되어 "아이 맛있어" 를 출력하고 break 를 통해 switch 문을 빠져 나가게 됩니다.

만약 변수 == 값2 라면 case 값1 은 실행되지 않고 case 값2 만 실행되게 됩니다.

또한 주의할 점으로는 '값' 에 위치하는 것들이 무조건 상수 이여야 한다는 것입니다. 만약 '값' 부분에 변수들이 오게된다면 오류가 발생하게 되는데 그 이유는 switch 문의 내부적인 처리 방법 때문입니다. (아래쪽 설명 되어 있습니다.)

마지막으로 switch 문의 default 는 if 문의 else 와 같은 역할을 합니다. 이도 저도 아닌 것들이 오는 case 이죠. 즉 위 예제의 경우 input 이 1 도 2 도 3 도 아닐 때 도달하는 경우가 됩니다.

그런데 위 switch 문에서 등장한 break 는 어디서 많이 본 것 같지 않습니까? 만약 그런 생각이 들었다면 당신은 C 언어 공부를 아주 충실히 하고 있다고 생각 됩니다. (만약 잘 모르겠다면 여기를

클릭하세요) break; 문을 실행하면 아래의 모든 case 들을 무시하고 switch 밖으로 빠져나가기 때문에 밥을 주었는데 강아지가 '아이 맛있어' 라고 할 일은 없게 됩니다.

하지만 만약 여러분이 break; 문을 빠뜨리게 되면 위와 같은 상황이 벌어질 수 있습니다.

```c
/* 실패작 */
#include <stdio.h>
int main() {
  int input;

  printf("마이펫 업그레이드\n");
  printf("무엇을 하실 것인지 입력하세요 \n");
  printf("1. 밥주기 \n");
  printf("2. 씻기기 \n");
  printf("3. 재우기 \n");

  scanf("%d", &input);

  switch (input) {
    case 1:
      printf("아이 맛있어 \n");

    case 2:
      printf("아이 시원해 \n");

    case 3:
      printf("zzz \n");

    default:
      printf("무슨 명령인지 못 알아 듣겠어. 왈왈 \n");
  }

  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
마이펫 업그레이드
무엇을 하실 것인지 입력하세요
1. 밥주기
2. 씻기기
3. 재우기
1
아이 맛있어
```

```
아이 시원해
zzz
무슨 명령인지 못 알아 듣겠어. 왈왈
```

와 같이 웃지 않을 수 없는 상황이 벌어집니다. 여러분들이 1 을 입력한다면 `case 1:` 이 실행되어 그 내용들이 모두 실행되지만 `break` 문으로 `switch` 문을 빠져 나가지 못해서 아래 `case` 들 까지 줄줄이 실행되어 위와 같은 꼴을 볼 수 있습니다.

```c
/* 영어 말하기 */
#include <stdio.h>
int main() {
  char input;

  printf("(소문자) 알파벳 읽기\n");
  printf("알파벳 : ");

  scanf("%c", &input);

  switch (input) {
    case 'a':
      printf("에이 \n");
      break;

    case 'b':
      printf("비 \n");
      break;

    case 'c':
      printf("씨 \n");
      break;

    default:
      printf("죄송해요.. 머리가 나빠서 못 읽어요  \n");
      break;
  }

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
(소문자) 알파벳 읽기
```

```
알파벳 : b
비
```

와 같이 나옵니다.

사실, 여기에 의문이 드는 사람들도 있습니다. 아까 위에서 *switch* 문은 정수 데이터만 처리한다고 했는데 왜 여기서는 문자 데이터도 처리가 되는 것인가?

그런데, 안타깝게도 이러한 의문이 5 초 이내로 해결되지 않으면 아마 앞에서 배운 내용을 까먹으셨을 것입니다. (그 내용을 보려면 여기를 클릭하세요) 왜냐하면 컴퓨터는 문자와 숫자를 구분 못합니다. 컴퓨터는 문자를 모두 숫자로 처리한 뒤, 우리에게 보여줄 때 에만 문자로 보여주는 것이지요. 따라서, 문자 = 정수 라고 생각해도 거의 무방합니다.

이쯤 switch 문을 배우고 나면 드는 의문이 하나 있습니다.

"정말로 switch 문이 우리에게 필요한가? if - else 로 다 해결되는데 왜 귀찮게 switch 문을 만들었을까? 차이는 단지 겉으로 얼마나 깔끔한지가 다를 뿐인데... 내부적으로 switch 문과 if-else 와는 차이가 없나요?"

정말로, 훌륭한 생각이라고 생각합니다. 위 질문에 대한 답변을 정확하게 이해하려면 어셈블리어에 대한 이해가 필요로 합니다.(참고로 if 문과 switch 문의 차이에 대한 설명을 자세하게 잘 다루는 곳)

위에 링크 걸은 사이트에 들어가 내용을 모조리 이해한다면 더할 나위 없이 좋겠으나 아마 C 언어를 처음 배우는 사람들의 경우 거의 이해를 못할 것이니 제가 간단하게 설명 드리겠습니다. (만약 아래의 내용을 이해하지 못하더라고 그냥 넘어가세요. 사실 어셈블리어를 배우지 않은 이상 이해하기 힘듭니다)

```
switch(input)
{
case 1:
    printf("아이 맛있어 ₩n");
    break;

case 2:
    printf("아이 시원해 ₩n");
    break;

case 3:
    printf("zzz ₩n");
    break;
default :
    printf("무슨 명령이야?");
    break;
}
return 0;
```

switch 문 이용!

```
if(input == 1)
{
    printf("아이 맛있어 ₩n");
}
else if(input == 2)
{
    printf("아이 시원해 ₩n");
}
else if(input == 3)
{
    printf("zzz ₩n");
}
else
{
    printf("무슨 명령이야?₩n");
}
```

위 두 그림은 같은 소스 코드를 switch 문과 if 문을 이용하여 나타난 것입니다. 사실, 외형적으로 동작하는 것은 차이가 없습니다. 단지 내부적으로 어떻게 처리되냐가 다를 뿐이지요.

일단 if 문의 경우 각 경우 마다 값들을 비교 합니다. 위 경우 값을 3 번 비교하겠네요. 왜냐하면 if 가 1 번, else if 가 2 번이고 else 의 경우 값의 비교 없이 자동으로 처리되는 것이므로 총 3 번 비교하게 됩니다. 즉, if 문을 이용하면 각 case 의 경우 비교하게 되므로 최악의 경우 모든 case 에 대해 값을 비교하는 연산 (어셈블리어에서는 CMP 연산을 합니다.) 을 시행하게 됩니다.

그런데 switch 문은 사뭇 다릅니다. switch 의 경우 내부적으로 jump table 이라는 것을 생성합니다. 이 때, jump table 의 크기는 case 의 값들에 따라 달라지는데, 예를 들어서 어떤 switch 문의 경우 case 1: ~ case 10: 까지 있었다고 합시다. 그렇다면 jump table 에는 값들이 0 부터 9 까지 들어가게 됩니다. 여기서 우리는 왜 case 값: 할 때, '값' 부분에 변수가 위치하면 안되는지 알게 됩니다. jump table 은 프로그램 초기에 작성 되기 때문에 이미 switch 문이 실행되기 전에 jump table 이 작성되게 됩니다. 따라서, '값' 부분에 변수가 들어가게 되면 jump table 에 무엇이 올지 알 수 없으므로 변수를 사용하면 안되는 것입니다.

이 값들은 무엇을 의미하냐면 각 case 별로 명령들이 위치한 곳의 주소를 가리키는데 예를 들어서 1 인 지점으로 점프하게 되면 "아이 시원해" 가 나오고 0 인 지점으로 점프하게 되면 "아이 맛있어" 라고 출력하라는 내용의 명령문들이 나옵니다. 이제, 변수의 값에 따라 변수가 3 이라면 jump table 의 3 번째 원소를 찾아서 그 값에 해당하는 곳으로 점프하게 됩니다.

(실제로 switch 문이 처리되는 과정은 이보다 약간 더 복잡하지만 어셈블리어를 배우지 않은 현재 상황으로써는 최선이라 생각됩니다)

따라서, switch 문을 이용하면 case 에 따라 CMP 연산이 늘어나는 것이 아니라 jump table 의 크기만 커질 뿐 성능에 있어서는 전혀 영향을 받지 않게 됩니다.

결론적으로 이야기 하자면 switch 문이 효과적으로 처리되기 위해서는 case 의 '값' 들의 크기가 그다지 크지 않아야 하고, '값' 들이 순차적으로 정렬되어 있고, 그 '값' 끼리의 차이가 크지 않다면 최고로 효율적인 switch 문을 이용할 수 있게 됩니다.

생각해 보기

문제 1

`switch` 문의 '값' 부분에 왜 정수만 와야 되는지 아십니까?(난이도 : 中上)

문제 2

앞서, `switch` 문이 내부적으로 처리 되는 부분에서 `case 1:` ~ `case 10:` 일 때 만 생각하였는데, 만약 `case 1:, case 3:, case 4:, case 10:` 과 같이 불규칙 적으로 `switch` 문이 적용된다면 컴퓨터는 `jump table` 를 어떻게 작성할까요 (난이도 : 最上)

뭘 배웠지?

- 어떤 정수 변수에 대해서 반복적으로 사용하는 `if-else` 문이 있다면 `switch` 를 사용하면 더 깔끔하게 바꿀 수 있습니다.
- 각 `case` 문 안에서 적절히 `break` 하는 것을 빠뜨리면 안됩니다.
- `default` 를 사용하면 `else` 문과 같은 효과를 낼 수 있습니다.

8

형 변환 (타입 캐스팅)

안녕하세요, 여러분! 이제 드디어 10 번째 강좌에 도달하였습니다. 아마 여태까지 느릿 느릿 진행되는 강좌를 꾸준히 기다리며 읽어와준 여러분들께 감사의 말을 전하고 싶습니다.

아마 잘 알고 있는 내용이지만 C 언어에서 각 변수들에는 고유의 **형(type)** 이 있습니다. 예를 들어서, `int a;` 로 선언된 변수 a 의 형은 `int` 형이고, `char b;` 로 선언된 변수 b 의 형은 `char` 형 입니다.[1] 또한 `float c;` 로 선언된 변수 c 의 형은 `float` 이고 `double d;` 로 선언된 변수 d 의 형은 `double` 이겠죠.

그런데 가끔씩 프로그래밍을 하다 보면 형이 다른 변수 끼리 대입을 하는 연산이 필요로 하게 됩니다. 예를 들어서 `double` 형 변수의 값을 `int` 형 변수에 대입하거나, `float` 형 변수에 `double` 형 변수의 값을 대입하는 것 등등 말이죠.

하지만 안타까운 사실은 형이 다른 변수 끼리의 대입이나 연산들이 모두 불법 이라는 것 입니다. 이건 마치 우리나라에서 달러로 물건을 구매하는 것과 똑같은 것이지요.. 그렇다면 어떻게 해야 할까요?

일단, 위 조건을 무시한 아래의 예제를 살펴 봅시다.

```c
/* 무시 */
#include <stdio.h>
int main() {
  int a;
  double b;

  b = 2.4;
  a = b;

  printf("%d", a);
  return 0;
}
```

1) char 을 어떻게 읽는지는 사람 마다 다른데, 보통 캐릭터 라고 하거나, 그냥 발음 그대로 찰 이라고도 합니다.

성공적(?) 으로 컴파일 한다면 아래와 같은 모습을 볼 수 있습니다.

실행 결과

```
2
```

어라, 아무런 애러도 없이 결과가 떡 하니 출력되었지만 눈썰미가 좋은 사람들은 `Output` 에 아래와 같은 메세지가 출력되었음을 알 수 있습니다.

```
: warning C4244: '=' : conversion from 'double' to 'int', possible loss of data
```

대충 직역해 보면 아래와 같은 의미 입니다.

컴파일 오류

```
경고 C4244 : '=' ": 'double' 로 부터 'int' 로의 형 변환, 데이터의 손실이 예상됨.
```

아마도 우리가 처음 보게 되었을 **컴파일러 경고(Warning) 메세지** 입니다. 똑똑한 컴퓨터는 우리가 `int` 형 변수에 `double` 형 변수의 값을 대입했다고 이야기 하고 있습니다. 또한, 데이터의 손실이 발생하게 된다고 귀띔까지 해주고 있습니다.

실제로, 결과를 확인해보면 데이터 손실이 발생하였음을 알 수 있습니다. 실행 결과를 보게 된다면 분명히 `a` 에 `2.4` 를 대입하였지만 `a` 의 결과는 2 로 나옵니다. (물론 `%d` 를 통해 정수 부분만 출력하게 해서 그렇다고 주장하는 사람들이 있는데 그렇다면 `%f` 로 바꿔서 출력해 보세요. 더 이상한 결과가 나올 것 입니다!)

`int` 형 변수에 (당연하게도, 3, 4 강을 제대로 배운 사람이라면 알겠지만) `double` 형 변수를 대입하면 소수 부분이 잘려서 정수 부분만 들어가게 됩니다. 이는 각 변수들이 메모리 상에 저장되는 특징이 다르기 때문이죠. 왜냐하면 `int` 형 변수는 처음 정의되는 시작 부터 메모리 상에 오직 정수 데이터만 받아 들이도록 설계되기 때문이죠.

그렇다면 훌륭한 학생이라면 여기서 의문이 생기게 됩니다.

> 도대체 컴퓨터는 실수를 어떻게 표현하는 거야!

컴퓨터가 실수를 표현하는 원리

주의 사항

`float` 이나 `double` 을 쓴다고 해서 꼭 이들 내부에서 어떠한 방식으로 실수가 표현되는지 알 필요는 없습니다. 하지만 한 번쯤 알면 재미있는 주제이니 궁금하신 분들은 쭉 읽어 보시고 이해가 잘 안되시는 분들은 그냥 넘어가셔도 좋습니다.

모두가 알고 있듯이 컴퓨터는 이진수로 모든 데이터를 표현합니다. 이전 강좌에서 컴퓨터가 어떠한 방식으로 이진수를 통해 양의 정수를 표현하는지 다루었고 이 강좌 에서는 음의 정수까지 어떻게 표현되는지 다루었습니다.

여기에서는 컴퓨터가 어떠한 방식으로 실수를 표현하는지에 대해 살펴볼 것입니다. 흔히 C 에서 실수를 보관하는 데이터 타입으로 `float` 과 `double` 을 들 수 있는데, 이들 데이터 타입이 실수를 어떠한 방식으로 보관하고 있는지 알아봅시다.

컴퓨터 상에서 실수를 표현하는 방법은 대표적으로 두 가지 방식을 들 수 있는데 하나는 **고정 소수점 (Fixed Point)** 방식이고 다른 하나는 **부동 소수점(Floating Point)** 방식 입니다. 눈치가 빠르신 분들은 `float` 타입의 `float` 이 어디서 온 것인지 알아채셨겠죠.

여러분이 사용하시는 대부분의 컴퓨터의 경우 아마 99.9% 부동 소수점 방식을 통해 실수를 표현하고 있을 것입니다. 그 이유가 고정 소수점 방식과 비교했을 때 같은 수의 비트만 사용해서 표현할 수 있는 수의 범위가 더 넓기 때문입니다.[2)]

이렇게 부동 소수점 방식을 통해 수를 표현하는 방법은 국제전기전자기술자협회(IEEE) 에서 1985 년에 **IEEE-754** 라는 이름으로 표준화 하였습니다.

IEEE 754

보통 우리가 수를 표현하는 방법은 아래와 같습니다.

$$123, 1234.123, -234$$

이 수는 아래와 같이 동일하게 표현할 수 있습니다. 아래 방식을 **과학적 표기(scientific notation)** 라고 부릅니다.

$$1.23 \times 10^2, 1.234123 \times 10^{-2}, -2.34 \times 10^2$$

2) 여기서 범위가 넓다는 말은 말 그대로 표현할 수 있는 가장 작은 수와 가장 큰 수의 차이가 더 크다는 뜻입니다. 대신 부동 소수점 방식은 고정 소수점 방식에 비해서 정밀도가 떨어집니다.

제가 중학교 때 위 사실을 배웠을 때 에는 저게 뭐에 쓸모 있는거지? 라고 생각 되었지만, 사실 위는 컴퓨터 상에서 실수를 표현하는 아주 중요한 기법 입니다.[3)]

마찬가지로 컴퓨터 상에서도 소수를 다음과 같이 표현합니다.

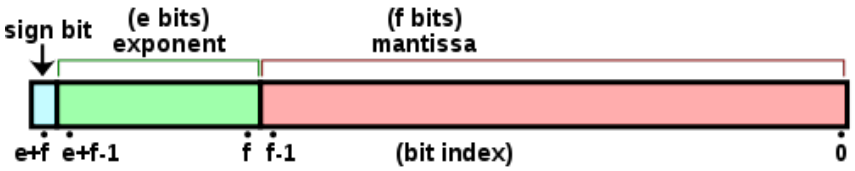
$$\pm f \times b^e$$

이 때, `f` 는 가수, `b` 는 밑, `e` 는 지수 입니다. 예를 들어서 123 의 경우 `f` 는 1.23, `b` 는 10, `e` 는 2 가 됩니다.

컴퓨터 상에서는 이진체계를 이용하기 때문에 `b` 의 값은 2 로 고정이 되어 있습니다.

따라서 소수 데이터를 보관할 때 `f, e` 의 값만 저장하면 됩니다. 그리고 맨 앞에 부호 비트를 위해서 1 비트 더 쓰게 됩니다.[4)] 부호 비트의 값이 0 이면 양수이고, 1 이면 음수가 됩니다.

아래 그림은 `IEEE 754` 에서 정의한 부동 소수점 표현 입니다.



우리가 자주 쓰는 `float` 의 경우 가수 부분이 23 비트를 차지하고, 지수 부분이 8 비트, 그리고 부호 비트가 1 비트를 차지하여 총 4 바이트를 차지하게 됩니다.

한편 `double` 의 경우 가수 부분이 52 비트고 지수 부분이 11 비트로 무려 8 바이트가 차지하는 거대 자료형 입니다.[5)]

이제 본격적으로 메모리 상에 실수가 어떻게 저장되는지 알아보기 위해 이진법으로 표현된 실수들을 십진법으로 바꾸고, 십진법으로 표현된 실수를 어떻게 이진법으로 바꾸는지 살펴봅시다.

소수의 10 진법 - 2 진법 진법 변환

먼저, 이진법으로 표시된 소수를 한 번 십진법을 바꾸어 보는 연습을 해봅시다.

$$10010.1011_{(2)}$$

소수점 이하 부분은 마찬가지로 자리수 마다 2^{-1}, 2^{-2} 순으로 쭉쭉 내려갑니다. 이는 10 진법 체계에서 10^{-1}, 10^{-2} 로 내려가는 것과 동일합니다. 따라서

3) 요즘에도 교육과정에 있는 것인지는 모르겠습니다..

4) 2 의 보수 표현법하고 살짝 다르죠!

5) double 이 double 인 이유는 간단하게도 float 보다 그냥 2 배 크기 때문에 그런 이름이 붙었습니다. 영어로도 배정밀도(double precision)라고 부릅니다

$$10010.1011_{(2)} = 2^4 + 2^1 + 2^{-1} + 2^{-3} + 2^{-4} = 18 + 0.5 + 0.125 + 0.0625 = 18.6875$$

와 같이 됩니다.

2 진법 으로 표시된 모든 소수들은 모두 십진법으로 변환이 가능합니다. 그렇다면 십진법 소수도 과연 이진법으로 바꿀 수 있을까요? 이번에는 -118.625를 한 번 이진소수로 바꾸어 봅시다.

$$-118.625 = -1110110_{(2)} - 0.625 = -1110110_{(2)} - 2^{-1} - 2^{-3} = -1110110.101_{(2)}$$

비슷한 방법으로 십진법으로 표시된 숫자들도 이진소수로 바꿀 수 있습니다.

그런데 안타까운 사실은 모든 10 진법으로 표현된 수는 2 진법으로 변환할 수 없습니다. 예를 들어 0.1 을 한 번 이진법으로 바꾸어 보세요. 10 진법으로는 딱 소수점 한 자리 만으로 표현이 가능하지만, 이진법으로 바꾼다면 아래와 같이 무한 소수가 나타나게 됩니다.

$$0.1 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \cdots = 0.0001100110011..._{(2)}$$

믿기지 않는 분들은 무한 등비수열의 합을 구하는 방법을 안다면 `0.1` 이 바뀐 무한 이진소수가 참임을 알 수 있습니다.

$$0.0001100110011..._{(2)} = \frac{\frac{1}{2^4}}{1 - \frac{1}{2^4}} + \frac{\frac{1}{2^5}}{1 - \frac{1}{2^4}} = \frac{1}{15} + \frac{1}{30} = \frac{1}{10}$$

컴퓨터는 이렇게 무한히 길게 나타나는 무한 소수들을 모두 메모리에 나타낼 수 없기 때문에 일정 부분만 잘라서 메모리에 보관하게 됩니다. 따라서 필연적으로 오차가 발생하게 됩니다.

IEEE 754 방식으로 소수 저장하기

자 그러면 이제 IEEE 754 방식 하에서 소수가 어떠한 방식으로 저장되는지 살펴봅시다.

가장 먼저 부호 비트에는 0 이상이면 0 이, 아니라면 1 이 할당됩니다. 앞서, `-118.625` 의 경우 부호 비트에 1 이 들어가겠죠?

두 번째로 변환된 이진수를 정규화(Normalization) 합니다. 정규화란, 어떠한 이진수를 `1.xxxx` 꼴로 만드는 것입니다. `-118.625` 의 경우, 이진수 형태인 `1110110.101` 을 `1.110110101` 로 바꾸는 것 입니다. 그렇다면 가수 부분에는 `xxxx` 부분, 즉 110110101 만 저장이 되겠지요.

이 때, 정규화 작업 시 얼마만큼 쉬프트 연산이 일어났는지 계산하여 지수 부분에는 얼마가 와야 되는지 알게 됩니다. 위의 경우 `1110110.101` 을 `1.110110101` 로 바꾸었으므로 쉬프트 연산이 6번 오른쪽으로 일어나게 되어서 지수에는 6 이 오게 됩니다.

`0.1` 처럼 무한 소수로 표현되는 수들의 경우 반올림을 하게 됩니다. 예를 들어 `0.1 = 0.00011001 10011001 10011001 10011001 ..` 로 나가는데, `float` 에 대입한다고 하면 `float` 의 가수 부분이 23 비트이므로 24 번째 비트에서 반올림을 하게 됩니다. 따라서, `0.1` 은 컴퓨터 상에 `0.00011001100110011001101` 로 보관 됩니다.

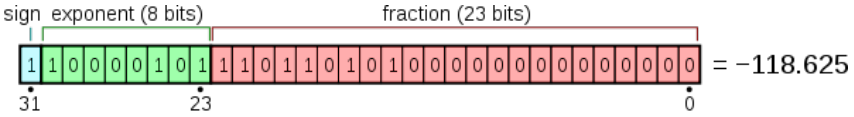
마지막으로 위에서 계산한 지수에 바이어스(Bias) 처리를 해줍니다. 이는 그냥 지수에 $2^{e-1}-1$ 만큼을 더해준다는 뜻입니다. 이 때, `e` 의 값은 지수 부분의 비트 수로, `float` 이면 8 이므로 127, `double` 형이면 11 이므로 1023 을 더하게 됩니다.

왜 계산한 지수에 바이어스 처리를 해주냐면은, 지수가 언제나 양수가 아니기 때문입니다. `-118.625` 의 경우 정규화 시 지수가 `+6` 이였으나 다른 소수들의 경우, 예를 들어 `0.625` 는 이진수로 `0.101` 인데 정규화 시, 왼쪽으로 쉬프트가 1 번 되므로 지수가 음수(-1) 가 됩니다.

2 의 보수 표현법으로 배운 우리로써는 **그냥 그러면 정수 표현하듯이 2 의 보수표현법으로 지수를 나타내면 안되냐** 라고 물을 수 있는데, 무조건 양수로 값을 집어넣는 것이 컴퓨터 입장에서 크기를 비교하기가 수월하기 때문입니다.

아무튼 `float` 의 경우 지수에 들어가는 값의 범위가 1 부터 254 까지 이고, `double` 의 경우에는 1 부터 2046 까지 가능하게 됩니다. 이 말은 `float` 의 지수 부분이 2^{-126} 부터 2^{127} 까지 가능하다는 의미가 되겠습니다.

자 그렇다면 `-118.625` 의 경우 지수 부분에 `6 + (127) = 133` 이 들어가게 됩니다. 133 은 이진수로 `10000101` 이지요. 따라서, `float a = -118.625;` 를 한 변수 `a` 의 메모리 구조를 살펴보면 아래와 같습니다.



이 때, 훌륭한 학생이라면 의문이 드는 점이 있을 것 입니다.

> 위에서 `float` 형 변수를 이용하게 되면 지수가 1 부터 254 까지 처리가 된다고 하였는데, 8 비트로 처리할 수 있는 수의 범위가 `0` ~ `255` 까지 이지 않나요? 0 과 255 는 어디로 갔나요?

좋은 질문입니다. 0 과 255 가 포함되지 않는 이유는 IEEE 754 에서 아래와 같이 정상적이지 않는 수를 표현하기 위해서 다음과 같이 규칙을 정했기 때문입니다.

종류	지수부	가수부
비 정 상 수 (Denormalized number)	0	0 이 아님
무한대	2^e-1	0
수가 아님(NaN)	2^e-1	0 이 아님

참고로 각 수에 대해 설명을 하자면

비정상 수 (Denormalized number)

비정상 수의 경우 2^{-127} 보다도 작아서 지수 부분에 바이어스 처리를 해도 1 이상이 되지 않는 수들을 말합니다. 따라서 이 들의 경우 더이상 $1.() \times 2^{-127}$ 의 형태로 표현할 수 없습니다. 이 수들은 그 대신 $0.() \times 2^{-127}$ 의 형태로 해석됩니다.

무한대

부호 비트 덕분에 IEEE 754 방식으로 음의 무한대와 양의 무한대를 표현할 수 있습니다. 무한대는 연산 과정에서 표현할 수 있는 가장 큰 수 보다 더 큰 값이 들어간다면 자동으로 발생하게 됩니다.

```c
#include <stdio.h>

int main() {
  float a = 1. / 0.f;
  printf("a : %f \n", a);
  return 0;
}
```

성공적으로 컴파일 하였으면

a : inf

와 같이 진짜로 무한대로 출력됨을 알 수 있습니다.

수가 아님 (NaN)

마지막 부류는 바로 수가 아님(Not-a-Number) 인 녀석들 입니다. 얘네들은 아래와 같이 엄밀히 값을 정할 수 없는 연산 중에 발생합니다. 예를 들어 $\infty - \infty, -\infty + \infty, 0 \times \infty, 0 \div 0, \infty \div \infty$ 등이 있습니다.

형 변환 (캐스팅)

그렇다면 우리는 경고가 나오지 않게 대입을 할 수 없는가요? 물론 있습니다. 서로의 형을 맞추어 버리면 되죠.

```c
/* 형변환 */
#include <stdio.h>
int main() {
  int a;
  double b;
```

```
  b = 2.4;
  a = (int)b;

  printf("%d", a);
}
```

성공적으로 컴파일 하면 아무리 눈을 굴려보아도 오류 나 경고 따위는 눈을 씻고 찾을 수 없게 됩니다. 그래서, 부푼 마음에 실행을 해 보면...

실행 결과

```
2
```

결과는 아까와 같은 2 입니다.

하지만, 아까와 같은 경고 메세지는 출력이 되지 않았습니다. 왜 일까요? 그 이유는 바로 우리가 강제로 형변환(캐스팅) 을 하였기 때문입니다.

어떠한 변수의 형을 바꿀려면 아래와 같이 하면 됩니다

```
(바꾸려는 형) 변수 이름
```

예를 들어, 위의 경우 `double` 로 선언된 `b` 를 `int` 로 바꾸었으므로 `(int)b` 라 하면 됩니다. 이 때, 형을 바꾼다는 것은 영구적으로 바뀌는 것이 아닙니다. 다시 말해 `double` 인 `b` 를 `int` 로 캐스팅 한다고 해도 `b` 가 `int` 인 변수가 되는 것이 아니라 계산식에서 일시적으로 `int` 형 변수로 바꾼 후 생각하라는 것 이죠. 즉,캐스팅을 하고도

```
printf("%f", b);
```

를 하게 되면 2.4 가 성공적으로 출력됩니다. 위 예제에서 우리는 강제로 형을 변환하였습니다. 따라서 컴파일러는 '아, 이 사람이 마음을 먹고 아예 형이 다른 변수들의 대입을 시도하는 구나' 라고 생각하고 오류 메세지를 출력하지 않게 되는 것 입니다.

```
/* 두 수의 비율 */
#include <stdio.h>
int main() {
  int a, b;
  float c, d;

  printf("두 숫자 입력 : ");
  scanf("%d %d", &a, &b);

  c = a / b;
```

```
  d = (float)a / b;

  printf("두 수의 비율 : %f %f", c, d);

  return 0;
}
```

성공적으로 컴파일 하면 (경고는 나오지만), 예를 들어 5 와 3 을 입력하였을 때 아래와 같이 나옵니다.

실행 결과

```
두 숫자 입력 : 5 3
두 수의 비율 : 1.000000 1.666667
```

와우! 신기하네요. 단지 형변환을 하고 안하고의 차이였지만 두 수의 비율이 하나는 정확하게 나오고 다른 하나는 부정확하게 나오는 군요. 일단, 위 예제에서 관건이 되는 부분은 바로 이 부분입니다.

```
c = a / b;
d = (float)a / b;
```

c 에는 a 를 b 로 나눈 값이 들어갑니다. d 에도 마찬가지인데 한 가지 차이점은 d 에서는 a 를 float 변수로 생각해서 계산하라라고 캐스팅 하였습니다. 이 때, 우리가 주목해야 하는 부분은 바로 a 와 b 가 정수형 변수라는 것 입니다.

컴퓨터에서 a/b 는 2 가지의 의미를 가집니다. 만약 a 와 b 중 어느 하나가 실수형 변수(float, double) 이라면 이는 정말 우리가 하는 나눗셈을 수행하게 됩니다. 다시말해 5/3 = 1.666666666666666666 이 되는 것 이죠. 하지만 a 와 b 가 모두 정수형 변수(char, int, long) 라면 컴퓨터는 위와 같은 나눗셈 연산을 수행하지 않고 소위 말하는 '몫' 을 계산하게 됩니다. 따라서 5/3 = 1 이 되는 것이지요.

따라서, (float)a/b 를 하게 되면 컴퓨터가 a 를 실수형 변수로 생각해가 되므로 a/b 처럼 몫을 계산하지 않고 정말로 실수형 나눗셈을 수행하게 된다는 것입니다. 따라서 d 에는 1.6666 ... 이 성공적으로 들어갈 수 있게 됩니다.

어때요? 형변환 하나로 많은 결과가 달라지지 않습니까? 실제로 형변환은 C 언어에서 매우 중요한 부분 중 하나 입니다. 또한 쓰임새도 상당히 많은데, 주로 실수형 변수에서 정수 부분만 추출할 때 사용되기도 합니다.

예를 들어 double a; int b; 일 때, b = (int)a; 라 하게 되면 변수 a 의 정수 부분 데이터만 b 로 넘어가게 되죠. 물론 b = a 로 해도 컴파일러가 알아서 캐스팅을 해주지만 그렇게 된다면 다른 프로그래머가 보았을 때, 이 것이 실수 인건지, 고의로 한 건지 모르므로 오해의 소지가 있습니다.

마지막으로 여러분에게 재미있는 문제를 내 보도록 하겠습니다. www.winapi.co.kr 이라는 사이트에서 가져온 문제인데, 여러분도 한 번 풀어보세요

생각 해보기

문제 1

임의의 실수에서 소수점 이하 두자리수만 추출하여 정수형 변수에 대입하라. 예를들어 사용자로부터 입력받은 실수 f 가 12.3456이라면 34만 추출한다. 이때 반올림은 고려하지 않아도 상관없다. f 가 달러 단위의 화폐 액수라고할 때 센트 단위만 추출해내는 경우라고 생각하면 된다. 다음 ???? 자리에 적합한 연산식을 작성하는 문제이다.

```
printf("실수를 입력하시오 : ");
scanf("%f", &f);
i = ? ? ? ?
printf("i=%d\n", i);
```

이 문제의 핵심은 음수이거나 소수점 이하의 자리수가 없는 경우까지 잘 고려하여 항상 잘 동작하는 코드를 만드는것이다.

9

C 언어의 아파트 - 배열

안녕하세요, 여러분. 지난 강의에 내 주었던 마지막 문제는 모두 푸셨나요? 저는 기본적으로 자신이 내준 문제는 스스로 무슨 수를 써서라도 풀어야 한다라는 것이 원칙이지만, 댓글로 한에서 답안을 공개할 수 있으니 여러분의 답안을 댓글로 남겨주시면 고맙겠습니다. 댓글이 저를 귀찮게 한다는 생각 말고요, 과감하게 댓글을 남겨 주세요. 저는 오히려 여러분의 댓글을 기다리고 있는 사람입니다.

얼마전에 Psi 는 친구로 부터 프로그램을 하나 짜 달라는 요청을 받았습니다. 친절한 Psi 는 그 친구의 요청을 흔쾌히 승낙했죠. 그런데, 그 친구가 요청한 프로그램은 그다지 평범한 프로그램이 아니였습니다. 그 친구의 반에 30 명의 학생들이 있는데 각 학생들의 성적들을 입력받아서 평균보다 낮은 사람들의 번호 옆에 '불합격', 평균 이상의 사람들에게 '합격' 이라는 메세지 까지 출력하는 프로그램을 말입니다.

그래서 Psi 는 생각하였습니다.

'30 명의 학생들의 점수를 입력 받아서 평균 까지는 구할 수 있겠는데 말야. 각 학생의 점수들을 보관하기 위한 변수들이 필요하단 말이야. 학생이 4 명 이라면 편하겠지만 30 명이라면.. `a1, a2, a3, a4, a5, ..... a30` 까지 각 변수의 값들을 언제 다 입력 받지? 젠장할! 이거 완전히 '캐' 노가다 아닌가.

아무튼, 프로그래밍을 하다가 위와 같이 여러 개의 값을 동시해 보관할 필요성이 생기게 되었습니다. 이전의 경우 여러개의 값을 보관할 경우에, 그 개수 만큼의 변수가 필요했었지요. 하지만 위 경우처럼 이용해야 할 변수의 개수가 매우 많아지게 되면 어떨 까요?

만약 전국의 학생수에 해당하는 프로그램을 작성하려면 10만개 이상의 변수들이 필요하게 됩니다. 이는, 프로그래머가 아무리 'Ctrl+C, Ctrl+V' 신공이 뛰어나다고 해도 불가능한 일이겠지요.

따라서, C 언어에 **배열(Array)** 이라는 것이 등장하게 되었습니다. 배열은, 간단히 말하자면 변수들의 집합이라고 말할 수 있습니다. 예를 들어서 `int` 형 배열의 경우, `int` 형 변수들이 메모리 상에 여러개 할당 되어 있는 것이지요.

배열의 기초

```c
/* 배열 기초 */
#include <stdio.h>
int main() {
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  printf("Array 3 번째 원소 : %d \n ", arr[2]);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Array 3 번째 원소 : 3
```

와 같이 나오게 됩니다.

일단 여러분은 새로운 것들이 나왔기 때문에 당황하는 분들이 계실 수 도 있습니다. 또한, 여태까지 배운 것들도 이해하기 힘든데 이제 더욱 어려운 것이 나타났구나! 라고 생각하시는 분들도 계실 것입니다. 하지만, 다행이도 여러분이 배우실 것들은 정말 쉬운 내용 들입니다. 그런 걱정 하시지 말고 차분히 읽어 보시면 됩니다.

앞서, 배열에 대해 잠깐 소개를 하였는데 배열은 말그대로 **특정한 형(Type) 의 변수들의 집합** 입니다. 변수를 정의할 때 에는

(변수의 형) (변수의 이름);

과 같이 정의했는데 배열은 그와 비슷하게도

(배열의 형) (배열의 이름)[원소 개수];

와 같이 해주면 됩니다. 위의 경우

```c
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

와 같이 해주었으므로, *int 형의 10 개의 원소를 가지는 배열 arr!* 이라고 생각하시면 됩니다. 다시말해, 이 배열은 10 개의 int 형 변수들을 보관 할 수 있게 됩니다. 만약 char arr[10] 이라 한다면 각 원소들이 모두 char 형으로 선언 됩니다. 또한 위와 같이 중괄호로 감싸 주었을

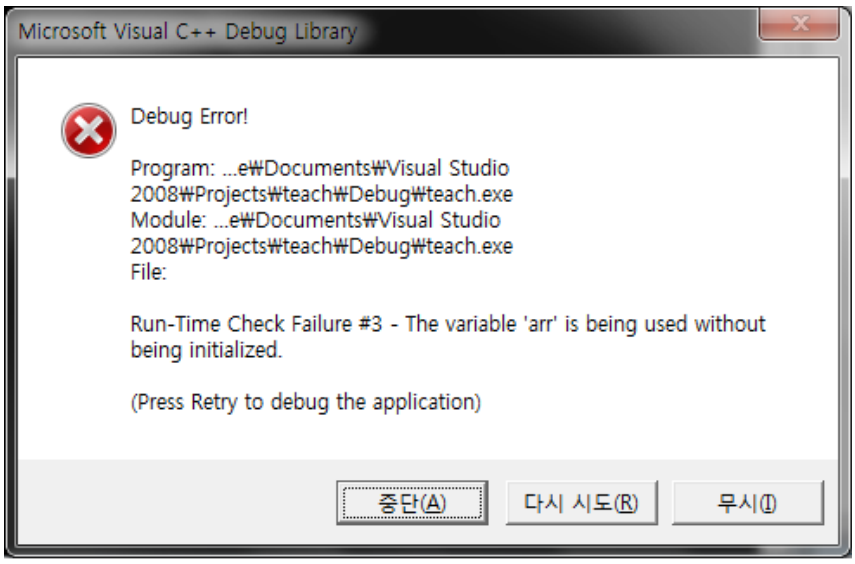
때, 배열의 각각의 원소에는 중괄호 속의 각 값들이 순차적으로 들어가게 됩니다. 배열의 첫 번째 원소에는 1, 두 번째 원소에는 2, ... 10 번째 원소에는 10 이 들어가게 됩니다.

그런데, 막연하게 배열을 정의해 놓고 보니 배열의 각각의 원소들에 접근하는 방법을 알려주지 않았군요. 사실 이는 간단합니다. 배열의 n 번째 원소의 접근하기 위해서는 arr[n-1] 와 같이 써주시면 됩니다. 즉, 대괄호[] 안에 접근하고자 하는 원소의 (번째수 - 1) 을 써주면 되죠. 예를 들어서

```
printf("Array 3 번째 원소 : %d \n ", arr[2]);
```

이라고 하면 배열의 3 번째 원소인 3 를 출력하게 됩니다. 많은 사람이 헷갈리는 부분인데, arr[2] 라고 하게되면 배열의 2 번째 원소인 2 를 출력하게 될 줄이라고 생각하지만 사실 3 번째 원소가 출력되게 됩니다. 즉, arr[0] 은 배열의 첫 번째 원소인 1 이 출력되고 arr[9] 는 배열의 10 번째 원소인 10 이 출력되게 됩니다.

만일 arr[10] 을 출력하려 한다면 무엇을 출력하게 될 까요? 한 번 해보세요. 아마도 아래와 같은 달콤한 애러 메세지를 볼 수 있게 될 것입니다.

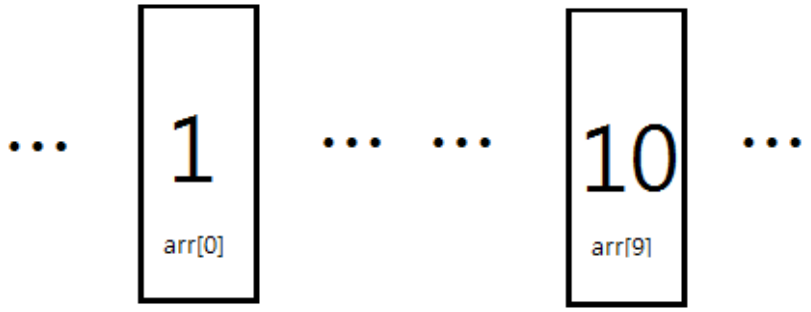


이는 배열의 존재하지도 않는 10 번째 원소를 참조하였기에 메모리 오류가 발생했다는 것 입니다. 사실, "배열의 10 번째 원소는 존재하지도 않으니, 우리가 참조한다면 0 과 같은 값들을 나타내면 되는데, 왜 위와 같이 껄렁한 오류 메세지를 출력하는 것인가?' 라고 생각할 것입니다. 그러기 위해선, 컴퓨터 상에 배열이라는 것이 어떻게 나타나게 되는이 알아야 합니다.

메모리 상에서의 배열

이미 3강에서도 다루었는데, 이 강의에서 저는 컴퓨터 메모리(RAM) 을 '직사각형 방이 쭉 나열되어 있는 형태' 로 표현하기로 했습니다. 사실 이는 다른 책들에서도 많이 이렇게 표현하므로 알아 두시는 것이 좋습니다. 이전 강의에서는 한 방의 크기가 1 바이트 였으므로 `int` 형의 경우 4 바이트를 차지하므로 배열의 한 원소를 표현하기 위해서는 4 칸을 차지해야 할 것입니다. 하지만, 각 4 개의 방을 모두 그리는 것이 힘들어서 메모리 한 칸을 그냥 4 바이트라고 합시다. 이 때, 위 배열 `arr` 의 경우 원소의 개수가 10 개 이므로 10 칸의 방을 차지하게 됩니다. (아쉽게도 그림 상에서는 모두 표현하지 못했으나 '…' 로 표현해 여러분의 상상력을 자극하고 있습니다 ㅎㅎ)

그런데, 메모리에 위 배열 하나만이 저장되어 있는 것일까요? 물론 아닙니다. 이 프로그램을 실행하는데 조차 수천개의 변수들의 메모리 상에 적재되어 이리저리 사라지고 저장되고 있습니다. 따라서 위 그림을 좀더 사실적으로 그리자면 아래와 같습니다.



즉, `arr[0]` 의 앞과 `arr[9]` 뒤에는 아무것도 없는 것이 아니라 다른 변수의 값들이 저장되고 있다는 사실 입니다. (물론 중간의 … … 에서는 `arr[1]` 부터 `arr[8]` 까지 '연속적' 으로 놓여있습니다.) 따라서, 훌륭한 운영체제라면 초짜 프로그래머가 다른 변수의 값들을 침범하는 것을 막기

위해 허락되지 않는 접근이 감지된다면 위와 같이 Run-Time 오류를 발생시키는 것이 정석입니다. 생각해 보세요. 당신이 스타를 하는데 미네랄이 갑자기 1000 에서 4 로 줄어든다면 기분이 어떻겠습니까?

배열 가지고 놀기

이제 본격적으로 배열을 가지고 놀아 보기로 합시다.

```c
/* 배열 출력하기 */
#include <stdio.h>
int main() {
  int arr[10] = {2, 10, 30, 21, 34, 23, 53, 21, 9, 1};
  int i;
  for (i = 0; i < 10; i++) {
    printf("배열의 %d 번째 원소 : %d \n", i + 1, arr[i]);
  }
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
배열의 1 번째 원소 : 2
배열의 2 번째 원소 : 10
배열의 3 번째 원소 : 30
배열의 4 번째 원소 : 21
배열의 5 번째 원소 : 34
배열의 6 번째 원소 : 23
배열의 7 번째 원소 : 53
배열의 8 번째 원소 : 21
배열의 9 번째 원소 : 9
배열의 10 번째 원소 : 1
```

아마, 여태까지 배운 내용을 잘 숙지하셨다면 위 소스를 어려움 없이 이해 하셨을 것이라고 생각합니다.

```c
int arr[10] = {2, 10, 30, 21, 34, 23, 53, 21, 9, 1};
```

일단, 배열의 정의 부분. arr 라는 원소를 10 개 가지는 int 형 배열을 선언한다라는 뜻이지요. 이 때, 각 원소의 값들은 중괄호 속에 있는 값들이 순차적으로 들어가므로 2, 10, 30 ~ 9, 1 까지 들어가게 됩니다.

```
for (i = 0; i < 10; i++) {
  printf("배열의 %d 번째 원소 : %d \n", i + 1, arr[i]);
}
```

이제, 배열의 원소들의 값들을 출력하는 부분 입니다. 잘 아시다싶이, for 문은 i = 0 부터 9 까지 1 씩 증가하면서 대입하는데 각 경우의 배열의 i 번째 원소를 출력하게 되므로 위 처럼 배열의 원소들의 값들이 출력되게 됩니다. 다시 한 번 기억하세요! 배열의 n 번째 원소를 참조하려면 arr[n-1] 로 입력해야 합니다!! arr[n] 이 '절대로' 아닙니다!

여기서 우리는 배열의 장점을 알 수 있습니다. 위 프로그램을 배열 없이 작성한다고 생각해보세요. 우리는 10 개의 서로 다른 변수를 만들어서 각각을 출력하는 작업을 해야 했을 것 입니다. 아래 (더러운) 코드는 위 예제와 동일한 내용을 10 개의 변수를 잡아서 직접 출력한 것을 보여 줍니다.

```
/* 더러운 코드 */
#include <stdio.h>
int main() {
  int a, b, c, d, e, f, g, h, i, j;
  a = 2;
  b = 10;
  c = 30;
  d = 21;
  e = 34;
  f = 23;
  g = 53;
  h = 21;
  i = 9;
  j = 1;

  printf("1 째 값 : %d \n", a);
  printf("2 째 값 : %d \n", b);
  printf("3 째 값 : %d \n", c);
  printf("4 째 값 : %d \n", d);
  printf("5 째 값 : %d \n", e);
  printf("6 째 값 : %d \n", f);
  printf("7 째 값 : %d \n", g);
  printf("8 째 값 : %d \n", h);
  printf("9 째 값 : %d \n", i);
  printf("10 째 값 : %d \n", j);

  return 0;
}
```

만일 여기에서 10 개의 변수의 값을 각각 입력받는 부분이라도 추가하라면 마우스라도 움켜 쥐고 울 것입니다. 하지만 배열을 이용하면 간단히 끝내 버릴 수 있습니다. 아래 예제 처럼 말이지요.

```c
/* 평균 구하기*/
#include <stdio.h>
int main() {
  int arr[5];  // 성적을 저장하는 배열
  int i, ave = 0;

  for (i = 0; i < 5; i++)  // 학생들의 성적을 입력받는 부분
  {
    printf("%d 번째 학생의 성적은? ", i + 1);
    scanf("%d", &arr[i]);
  }
  for (i = 0; i < 5; i++)  // 전체 학생 성적의 합을 구하는 부분
  {
    ave = ave + arr[i];
  }

  // 평균이므로 5 로 나누어 준다.
  printf("전체 학생의 평균은 : %d \n",  ave / 5);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
1 번째 학생의 성적은? 100
2 번째 학생의 성적은? 90
3 번째 학생의 성적은? 80
4 번째 학생의 성적은? 70
5 번째 학생의 성적은? 60
전체 학생의 평균은 : 80
```

사실, 위 평균 구하는 프로그램은 앞에서도 한 번 만들어 보았는데 이번에는 배열을 이용하여 프로그램을 만들어 보았습니다. 이전보다 오히려 더 복잡해진 느낌이지만, 학생 개개인의 성적을 변수로 보관하기 때문에 더 많은 작업들을 할 수 있게 되죠.

```c
for (i = 0; i < 5; i++)  // 학생들의 성적을 입력받는 부분
{
  printf("%d 번째 학생의 성적은? ", i + 1);
  scanf("%d", &arr[i]);
}
```

위 소스에서 학생들의 성적을 입력받는 부분 입니다. 별다른 특징이 없습니다. 이전에 scanf 에서 &(변수명) 형태로 값을 입력 받았는데 배열도 마찬가지로 같습니다. (이 부분에 대해서는 나중에 좀 더 자세히 다루도록 하지요. 왜 &(arr[i]) 로 안해도 상관이 없는지...) 배열도, &arr[i] 로 쓰면 arr 배열의 (i+1) 번째 원소에 입력을 받게 됩니다. 이 때, arr 이 int 형 배열이기에 각 원소도 모두 int 형 이므로 %d 를 사용하게 되지요.

```
for (i = 0; i < 5; i++)  // 전체 학생 성적의 합을 구하는 부분
{
  ave = ave + arr[i];
}
```

이제 ave 에 배열의 각 원소들의 합을 구하는 부분 입니다. 배열도 변수의 모음이기에, 각 원소들은 모두 변수처럼 사용 가능합니다. 물론 배열의 각 원소들끼리의 연산도 가능합니다. 예를 들어서

```
ave[4] = ave[3] + ave[2] * i + ave[1]/i;
```

와 같이 해도 ave[4] 는 정확한 값이 들어가게 되지요. 마지막으로

```
printf("전체 학생의 평균은 : %d \n", ave / 5);
```

라 하면 전체 학생들의 평균을 구할 수 있게 됩니다.

자, 그렇다면 아까 위의 친구가 부탁했던 프로그램을 만들 수 있는 수준까지 도달할 수 있겠습니다. 한 번 여러분이 짜보고 제가 만든 소스코드와 비교해 보는 것도 좋은 방법 인 것 같습니다. (참고로 저는 학생을 30 명으로 하면 처음에 데이터 입력하기가 너무 힘들어서 그냥 10 명으로 했으니 여러분은 마음대로 하시기 바랍니다.)

```
/* 친구의 부탁 */
#include <stdio.h>
int main() {
  int arr[10];
  int i, ave = 0;
  for (i = 0; i < 10; i++) {
    printf("%d 번째 학생의 성적은? ", i + 1);
    scanf("%d", &arr[i]);
  }
  for (i = 0; i < 10; i++) {
    ave = ave + arr[i];
  }
  ave = ave / 10;
  printf("전체 학생의 평균은 : %d \n", ave);
  for (i = 0; i < 10; i++) {
    printf("학생 %d : ", i + 1);
    if (arr[i] >= ave)
```

```
      printf("합격 \n");
    else
      printf("불합격 \n");
  }
  return 0;
}
```

성공적으로 컴파일 하였다면 아래와 같은 화면을 볼 수 있을 것입니다.

실행 결과

```
1 번째 학생의 성적은? 30
2 번째 학생의 성적은? 90
3 번째 학생의 성적은? 80
4 번째 학생의 성적은? 68
5 번째 학생의 성적은? 99
6 번째 학생의 성적은? 100
7 번째 학생의 성적은? 78
8 번째 학생의 성적은? 23
9 번째 학생의 성적은? 85
10 번째 학생의 성적은? 49
전체 학생의 평균은 : 70
학생 1 : 불합격
학생 2 : 합격
학생 3 : 합격
학생 4 : 불합격
학생 5 : 합격
학생 6 : 합격
학생 7 : 합격
학생 8 : 불합격
학생 9 : 합격
학생 10 : 불합격
```

유후! 어때요. 잘 출력되는 지요.

```
for (i = 0; i < 10; i++) {
  printf("학생 %d : ", i + 1);
  if (arr[i] >= ave)
    printf("합격 \n");
  else
```

```
    printf("불합격 \n");
}
```

사실, 위 소스는 앞에서 보았던 우리의 평균 구하는 소스에 약간 더해서 만든 것 이므로 위 부분만 살펴 보면 되겠습니다. i 가 0 부터 9 까지 가면서 배열의 각 원소들을 ave 와 비교하고 있습니다. 만약, ave 이상이라면 합격, 그렇지 않다면 불합격을 출력하게 말이지요. 솔직히 이 정도 수준의 프로그램 소스는 이제 더이상 설명해 줄 필요가 없어진 것 같습니다. (저만 그런가요? 만약 그렇지 않다면 이전의 강의들을 다시 한 번 정독하기를 강력하게 권합니다)

소수 찾는 프로그램

이번에는 배열을 활용한 프로그램을 하나 더 살펴 보겠습니다. 이번 프로그램은 '배열' 을 활용한 소수 찾는 프로그램 입니다. **소수(prime number)**는 1 과 자신을 제외한 약수가 하나도 없는 수를 일컫습니다.

예를 들어, 2 와 3 은 소수 이지만 4 는 2 가 약수 이므로 소수가 아니지요. 또한 1 도 소수가 아닙니다. 아무튼, 소수를 찾는데 배열을 활용한다는 것은 이전에 찾은 소수들을 배열에 저장하여, 어떠한 수가 소수인지 판별하기 위해 그 수 이하의 소수들로 나누어 본다는 뜻입니다.

만일, 그 수 이하의 모든 소수들로 나누었는데 나누어 떨어지는 것이 없다면 그 수는 소수가 됩니다. 또한, 짝수 소수는 2 가 유일하므로 홀수들에 대해서만 계산하도록 합니다 . 또한, 짝수 소수는 2 가 유일하므로 홀수들에 대해서만 계산하도록 합니다.

이러한 아이디어를 바탕으로 프로그램을 짜 보겠습니다. 여러분은 아래 제가 구현한 코드를 보지 말고 한 번 스스로 해보시기 바랍니다. 참고로 저의 프로그램은 소수를 1000 개 만 찾습니다.

```
/* 소수 프로그램 */
#include <stdio.h>
int main() {
  /* 우리가 소수인지 판별하고 있는 수 */
  int guess = 5; /* 소수의 배열 */
  int prime[1000]; /*     현재까지 찾은 (소수의 개수 - 1)      아래 두 개의 소수를
                        미리 찾았으므로      초기값은 1 이 된다.      */
  int index = 1; /* for 문 변수 */
  int i;         /* 소수인지 판별위해 쓰이는 변수*/
  int ok;        /* 처음 두 소수는 특별한 경우로 친다 */
  prime[0] = 2;
  prime[1] = 3;
  for (;;) {
    ok = 0;
    for (i = 0; i <= index; i++) {
      if (guess % prime[i] != 0) {
        ok++;
      } else {
```

```
        break;
      }
    }
    if (ok == (index + 1)) {
      index++;
      prime[index] = guess;
      printf("소수 : %d \n", prime[index]);
      if (index == 999) break;
    }
    guess += 2;
  }
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
... (생략) ...
소수 : 7727
소수 : 7741
소수 : 7753
소수 : 7757
소수 : 7759
소수 : 7789
소수 : 7793
소수 : 7817
소수 : 7823
소수 : 7829
소수 : 7841
소수 : 7853
소수 : 7867
소수 : 7873
소수 : 7877
소수 : 7879
소수 : 7883
소수 : 7901
소수 : 7907
소수 : 7919
```

와 같이 소수가 쭉 나오는 것을 볼 수 있습니다. 일단 위 소스코드의 핵심적인 부분만 설명해 보도록 하겠습니다.

```
for (i = 0; i <= index; i++) {
  if (guess % prime[i] != 0) {
    ok++;
  } else {
    break;
  }
}
```

위 부분은 guess 이하의 모든 소수들로 나누어 보고 있는 작업 입니다. index 는 (배열에 저장된 소수의 개수 - 1) 인데 prime[i] 로 접근하고 있으므로 배열의 모든 소수들로 나누어 보게 됩니다.

만일 guess 가 prime[i] 로 나누어 떨어지지 않는다면 ok 를 1 증가 시킵니다. 그리고 나누어 떨어진다면 소수가 아니므로 바로 break 되서 루프를 빠져 나가게 됩니다. 만일 ok 가 prime 배열에 저장된 소수의 개수, 즉 (index + 1)과 같다면 자기 자신 미만의 모든 소수들로도 안 나누어 떨어진다는 뜻이 되므로 소수가 됩니다.

이 때, 주의해야 할 점은 한 개의 수를 검사할 때 마다 ok 가 0 으로 리셋되어야 합니다. 그렇지 않다면 정확한 결과를 얻을 수 없겠죠?

```
if (ok == (index + 1)) {
  index++;
  prime[index] = guess;
  printf("소수 : %d \n", prime[index]);
  if (index == 999) break;
}
```

따라서, 위와 같이 index 를 하나 더 증가시킨 후 prime[index] 에 guess 를 추가 시켜 줍니다. 만일 index 가 999 가 된다면 배열이 꽉 찼단 뜻이 되므로 break 를 해서 for(;;) 를 빠져 나가게 됩니다. 어때요? 배열을 이용하여 정말 많은 일을 할 수 있지요?

배열의 중요한 특징

만약 똑똑한 사람이라면 다음과 같이 생각할 수 있을 것 입니다.

> 처음에 배열의 원소의 수를 숫자로 지정하지 않고 변수로 지정해도 될까? 예를 들어 arr[i] 라던지 말이야. 그렇게 된다면 위 프로그램에서 반의 총 학생 수를 입력 받아서 딱 필요한 데이터만 집어 넣으면 되잖아?

그렇다면, 그의 아이디어를 빌려서 프로그램을 만들어 봅시다.

```c
/* 과연 될까? */
#include <stdio.h>
int main() {
  int total;
  printf("전체 학생수 : ");
  scanf("%d", &total);
  int arr[total];
  int i, ave = 0;

  for (i = 0; i < total; i++) {
    printf("%d 번째 학생의 성적은? ", i + 1);
    scanf("%d", &arr[i]);
  }
  for (i = 0; i < total; i++) {
    ave = ave + arr[i];
  }

  ave = ave / total;
  printf("전체 학생의 평균은 : %d \n", ave);

  for (i = 0; i < total; i++) {
    printf("학생 %d : ", i + 1);
    if (arr[i] >= ave)
      printf("합격 \n");
    else
      printf("불합격 \n");
  }

  return 0;
}
```

만약 컴파일 한다면 아래에 수많은 오류가 쏟아져 나오는 것을 볼 수 있습니다.

```
1>c:\Users\lee\Documents\Visual Studio 2008\Projects\teach\teach\a.c(13) : error C2109: 첨자는 배열 또는 포인터 형식을 사용해야 합니다.
1>c:\Users\lee\Documents\Visual Studio 2008\Projects\teach\teach\a.c(15) : error C2065: 'i' : 선언되지 않은 식별자입니다.
1>c:\Users\lee\Documents\Visual Studio 2008\Projects\teach\teach\a.c(15) : error C2065: 'i' : 선언되지 않은 식별자입니다.
1>c:\Users\lee\Documents\Visual Studio 2008\Projects\teach\teach\a.c(15) : error C2065: 'i' : 선언되지 않은 식별자입니다.
1>c:\Users\lee\Documents\Visual Studio 2008\Projects\teach\teach\a.c(17) : error C2065: 'ave' : 선언되지 않은 식별자입니다.
1>c:\Users\lee\Documents\Visual Studio 2008\Projects\teach\teach\a.c(17) : error C2065: 'ave' : 선언되지 않은 식별자입니다.
1>c:\Users\lee\Documents\Visual Studio 2008\Projects\teach\teach\a.c(17) : error C2065: 'arr' : 선언되지 않은 식별자입니다.
1>c:\Users\lee\Documents\Visual Studio 2008\Projects\teach\teach\a.c(17) : error C2065: 'i' : 선언되지 않은 식별자입니다.
```

왜 오류가 나올까? 라고 고민한다면 3강 맨 아래 부분을 다시 보시길 바랍니다.[1)]

왜냐하면 변수는 무조건 최상단에 선언되야 되기 때문입니다! 위와 같이 배열 `arr` 과 변수 `i, ave` 가 변수 선언문이 아닌 다른 문장 다음에 나타났으므로 C 컴파일러는 무조건 오류로 처리하게 됩니다. (물론 C++ 에서는 가능합니다)

아아. 애초에 사람이 입력하는 대로 배열의 크기를 임의로 정할 수 는 없는 것이였군요. 그렇다면, 그냥 변수 크기 지정시 특정한 값이 들어있는 변수가 가능한지 살펴 봅시다.

1) 2018년 현재, 더이상 변수를 최상단에 선언하지 않아도 됩니다. 아마 오류가 발생하지 않을 것입니다.

```c
/* 설마 이것도? */
#include <stdio.h>
int main() {
  int total = 3;
  int arr[total];
  int i, ave = 0;

  for (i = 0; i < total; i++) {
    printf("%d 번째 학생의 성적은? ", i + 1);
    scanf("%d", &arr[i]);
  }
  for (i = 0; i < total; i++) {
    ave = ave + arr[i];
  }

  ave = ave / total;
  printf("전체 학생의 평균은 : %d \n", ave);

  for (i = 0; i < total; i++) {
    printf("학생 %d : ", i + 1);
    if (arr[i] >= ave)
      printf("합격 \n");
    else
      printf("불합격 \n");
  }

  return 0;
}
```

과연 성공할까요?

```
1>컴파일하고 있습니다...
1>8.c
1>c:₩users₩lee₩documents₩visual studio 2008₩projects₩teach₩teach₩8.c(5) : error C2057: 상수 식이 필요합니다.
1>c:₩users₩lee₩documents₩visual studio 2008₩projects₩teach₩teach₩8.c(5) : error C2466: 상수 크기 0의 배열을 할당할 수 없습니다.
1>c:₩users₩lee₩documents₩visual studio 2008₩projects₩teach₩teach₩8.c(5) : error C2133: 'arr' : 알 수 없는 크기입니다.
1>빌드 로그가 "file://c:₩Users₩Lee₩Documents₩Visual Studio 2008₩Projects₩teach₩teach₩Debug₩BuildLog.htm"에 저장되었습니다.
1>teach - 오류: 3개, 경고: 0개
========== 빌드: 성공 0, 실패 1, 최신 0, 생략 0 ==========
```

아아. 역시 우리의 기대를 처절하게 저버리고 오류를 내뿜는 컴파일러.

이는 C 언어에 처음에 배열의 크기를 변수를 통해 정의할 수 없게 규정하고 있기 때문입니다. (사실, '동적 할당' 이라는 방법으로 억지로 해서 정의할 수 있으나 이 부분에 대한 이야기는 나중에 다루도록 합시다.) 왜냐하면 처음에 컴파일러가 배열을 처리할 대 메모리 상에 공간을 잡아야 하는데 이 때, 잡아야 되는 공간의 크기가 반드시 상수로 주어져야 하기 때문입니다. 지금 수준에서 깊게 설명하는 것은 너무 무리인 것 같으니 그냥 '배열의 크기는 변수로 지정할 수 없다' 정도로 넘어가도록 합시다.

상수 (Constant)

상수는 변수의 정반대로 처음 정의시 그 값이 바로 주어지고, 그 값이 영원히 바뀌지 않습니다.

```
/* 상수 */
#include <stdio.h>
int main() {
  const int a = 3;

  printf("%d", a);
  return 0;
}
```

만약 성공적으로 컴파일 하였다면

```
3
```

와 같이 나오게 됩니다. 상수는 아래와 같이 정의합니다.

const (상수의 형) (상수 이름) = (상수의 값);

위 소스의 경우, a 라는 이름의 int 형 상수이고 그 값은 3 이라는 것을 나타내고 있습니다.

```
const int a = 3;
```

상수라고 해서 꼭 특별한 것이 있는 것은 아닙니다. 단지, 처음에 한 번 저장된 값은 '절대로' 변하지 않는 다는 점일 뿐이지요. 그렇기 때문에 처음 상수를 정의시 값을 정의해 주지 않는다면

```
#include <stdio.h>
int main() {
  const int a;

  printf("%d", a);
  return 0;
}
```

와 같이 컴파일시 아래와 같이 나타나게 됩니다.

```
1>컴파일하고 있습니다...
1>a.c
1>c:₩users₩lee₩documents₩visual studio 2008₩projects₩teach₩teach₩a.c(6) : warning C4700: 초기화되지 않은 'a' 지역 변수를 사용했습니다.
1>링크하고 있습니다...
1>매니페스트를 포함하고 있습니다...
1>빌드 로그가 "file://c:₩Users₩Lee₩Documents₩Visual Studio 2008₩Projects₩teach₩teach₩Debug₩BuildLog.htm"에 저장되었습니다.
1>teach - 오류: 0개, 경고: 1개
========== 빌드: 성공 1, 실패 0, 최신 0, 생략 0 ==========
```

상수는 또한 그 특성 답게 그 값 자체를 바꿀 수 없습니다. 예를 들어서

```c
#include <stdio.h>
int main() {
  const int a = 2;

  a = a + 3;
  printf("%d", a);
  return 0;
}
```

를 한다면,

```
1>------ 빌드 시작: 프로젝트: teach, 구성: Debug Win32 ------
1>컴파일하고 있습니다...
1>a.c
1>c:₩users₩lee₩documents₩visual studio 2008₩projects₩teach₩teach₩a.c(6) : error C2166: l-value가 const 개체를 지정합니다.
1>빌드 로그가 "file://c:₩Users₩Lee₩Documents₩Visual Studio 2008₩Projects₩teach₩teach₩Debug₩BuildLog.htm"에 저장되었습니다.
1>teach - 오류: 1개, 경고: 0개
========== 빌드: 성공 0, 실패 1, 최신 0, 생략 0 ==========
```

와 같은 오류가 발생하게 됩니다. 즉, 상수는 어떠한 짓으로도 값을 변경할 수 없는 불멸의 데이터입니다. 이러한 특성 때문에 여러분은 아마 '배열의 크기를 상수로 지정할 수 는 없을까?' 라는 생각을 하게 됩니다. 하지만, 아래와 같은 코드를 보면 이러한 생각이 깨지게 되죠.

```c
#include <stdio.h>
int main() {
  int b = 3;
  const int a = b;
  char c[a];
  return 0;
}
```

즉, 이는 상수 a 로 배열의 크기가 할당이 가능하게 된다면 다시 말해 변수 b 의 크기로 배열의 크기를 지정할 수 있다는 말이 되기 때문에 이전에 '변수로 배열의 크기를 지정할 수 없다' 라는 사실에 모순됩니다.

컴퓨터 프로그래밍을 하다 보면 상수를 사용할 일이 꽤나 많습니다. 예를 들어서 계산기 프로그램을 만들 때 에는 Pi 의 값을 상수로 지정해 놓게 된다면 변수로 지정할 때 보다 훨씬 안전해지게 됩니다. 왜냐하면 변수로 Pi 의 값을 지정했을 때, 프로그래머가 코딩상의 실수로 그 값을 바꾼다면 찾아낼 도리가 없지만, 상수로 지정시에 그 값을 실수로 바꾸도록 코딩을 해도 애초에 컴파일러가 오류를 뿜기 때문에 오류를 미연에 방지할 수 있습니다.

초기화 되지 않은 값

우리가 변수의 값을 초기화 하지 않는다면 그 변수는 무슨 값을 가질 까? 라는 생각을 한 분들이 많을 것 같습니다. 0 을가질까요? 아닙니다. 0 도 값 이지 않습니까? 0 을 가진다면 0 이라는

값을 가진다는 것 이지요? 그렇다면 한 번 해보지요.값이 대입되지 않은 변수의 값을 출력해보는 프로그램을 짜보면 아래와 같습니다.

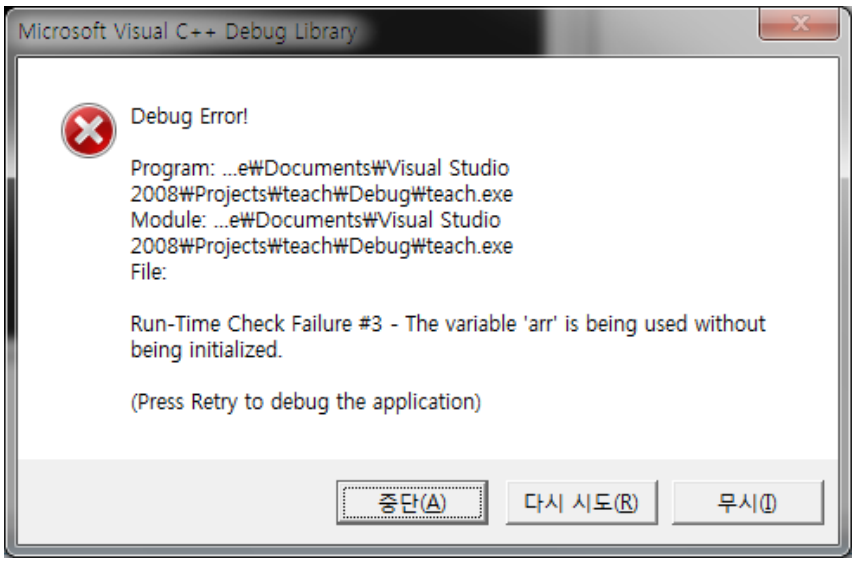
```
#include <stdio.h>
int main() {
  int arr;

  printf("니 값은 모니 : %d", arr);
  return 0;
}
```

컴파일 해보면 아래와 같은 경고를 볼 수 있습니다.

컴파일 오류

```
warning C4700: 초기화되지 않은 'arr' 지역 변수를 사용했습니다.
```

아마, 심상치 않지만 그래도 오류가 없으니 실행은 해볼 수 있겠군요. 아마 실행해 보면 아래와 같은 모습을 보실 수 있을 것 입니다.



아니 이럴수가! 변수 arr 의 값을 보기위해 값을 출력하려고 했더니만, 런타임 오류(프로그램 실행 중에 발생하는 오류)가발생하군요. 운영체제는 초기화 되지 않은 변수에 대한 접근 자체를 불허하고 있습니다. 이 때문에 우리는 이 변수에 들어있는 값을영영 보지 못하게 됩니다.

```
/* 초기화 되지 않은 값 */
#include <stdio.h>
int main() {
  int arr[3];
  arr[0] = 1;
```

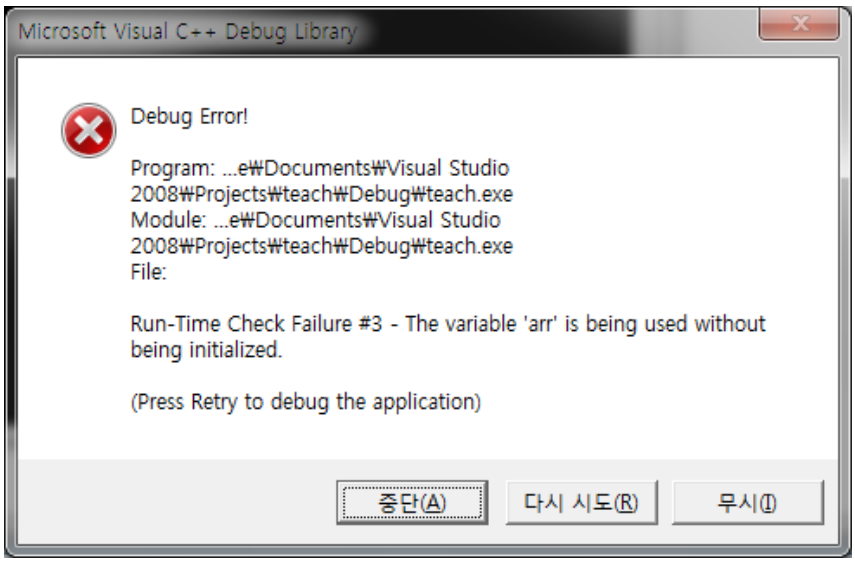
```
  printf("니 값은 모니 : %d", arr[1]);  // arr[0] 이 아닌 arr[1] 을 출력

  return 0;
}
```

이번에는 배열의 경우를 살펴 봅시다. 상콤한 기분으로 컴파일을 했으나,

컴파일 오류

`warning C4700: 초기화되지 않은 'arr' 지역 변수를 사용했습니다.`

와 같은 경고가 발생합니다. 심지어 앞 예제에서 발생했던 경고와 번호(C7400)와 동일하군요. 불안감에 사로잡혀 실행해보면.. 아니나 다를까 아래와 같은 오류 메세지 창을 보게 됩니다.



역시, `arr[1]` 의 값은 정의되어 있지 않기 때문에 이 값을 출력할 생각은 꿈도 꾸지 말라는 것이라는 것이죠. 참으로 야속한 컴퓨터 입니다.

```
/* 초기화 되지 않은 값 */
#include <stdio.h>
int main() {
  int arr[3] = {1};
  printf("니 값은 모니 : %d", arr[1]);

  return 0;
}
```

이번에는 마지막으로 거의 자포자기 한 심정으로 위 소스를 컴파일 해 봅시다.

실행 결과

```
니 값은 모니 : 0
```

오잉! 경고도, 오류도 나타나지 않습니다. 더군다나, 0 이라는 값이 출력되었습니다! 놀랍군요. 우리는 위 문장에서 `arr[1]` 에 0 을 집어 넣는 다는 말은 한 번이라도 하지 않았습니다. 사실 여러분은 위와 같이 배열을 정의한데에 놀랄 수도 있습니다. 배열의 원소는 3 개나 있는데 값은 오직 1 개 밖에 없기 때문이죠. 하지만 여러분들은 위 문장이 다음과 같은 역할을 한다는 것은 직감적으로 알 수 있습니다.

int arr[3]; arr[0] = 1;

물론 맞는 말 입니다. `printf` 부분을 `arr[0]` 출력으로 바꾸어 보면 여러분이 생각했던 대로 1 이 출력됩니다. 하지만, `arr[1]` 이 도대체 왜 오류가 나지 않는 것일까요? 아까전에 `int arr[3]; arr[0] = 1;` 방법으로 해서 끔찍한 오류가 발생되는 것을 여러분이 두 눈으로 톡톡히 보셨지 않습니까? 그 이유는

```
int arr[3] = {1};
```

와 같이 정의한다면 컴파일러가 내부적으로 아래와 같이 생각하기 때문입니다.

```
int arr[3] = {1, 0, 0};
```

따라서, 자동적으로 우리가 특별히 초기화 하지 않은 원소들에는 0 이 들어가게 됩니다.

그렇다면

```
int arr2[5] = {1, 2, 3};
```

은 어떻게 될까요? 역시, 해보면

```
int arr2[5] = {1, 2, 3, 0, 0}
```

과 같이 한 것과 똑같이 됩니다.

이상으로, 배열에 대한 첫 번째 강의를 마치도록 하겠습니다. 아직 여러분은 배열에 대해 모든것을 다 알고 계신 것은 아닙니다. 다음 강의에서는 더욱 놀라운 배열의 기능을 알아 보도록 합시다.

생각해 볼 문제

문제 1

위 입력받는 학생들의 성적을 높은 순으로 정렬하는 프로그램을 만들어 보세요.

문제 2

입력받은 학생들의 성적을 막대 그래프로 나타내는 프로그램을 만들어 보세요.

다차원 배열

안녕하세요, 여러분. 아마 이쯤 되면 여태까지 공부하셨던 사람들 중 일부는 아, 내 머리는 컴퓨터 언어를 배우기에 최적화 되어 있지 않나 보다 하고 포기하는 사람들과 오오... 내 맘대로 프로그램을 만들고 주물럭 주물럭 거릴 수 있는 것이 신기한데?? 하는 두 가지 부류의 사람들로 나뉘게 됩니다.

사실, 여기 까지 온 것 만으로도 정말 대단하다고 말 할 수 있습니다. 왜냐하면 인터넷을 통해 무언가 짬짬히 보아서 공부해 나가는 것은 쉬운 일이 아니기 때문이죠. 여러 게임의 유혹도 있고, 내가 이걸 배우는 시간에 채팅이나 하면 좋을 것을.. 와 같은 생각도 들기 때문이죠.

하지만, 저는 여러분이 조금만 더 힘을 내어 이러한 유혹을 이겨내고 C 언어의 끝에 도달하시기 바랍니다. 사실 배열과 앞으로 나오는 포인터 부분만 넘어간다면 더이상 어려울 부분이 없기 때문이죠. 그 부분이 넘어가 C 언어 끝에 도달하게 되면, 윈도우 API 를 공부하여 윈도우 앱도 만들고, Direct X 나 OpenGL 등을 공부해서 3D 게임 까지. 뿐만 아니라 소켓 프로그래밍을 공부한다면 친구들과 채팅할 수 있는 프로그램도 만들 수 있고 게임 서버들도 만들 수 있습니다.

C 언어를 배움으로써 얻을 수 있는 위 많은 것들을 쉽게 포기하실 것 입니까? 물론, 나가실 분은 조용히 뒤로가기를 누르셔도 상관 없습니다. 다만, 이 순간의 클릭이 당신의 앞날을 좌우 할 지도 모른다는 사실을 잊지 마세요. 그리고 참, 이전 내용이 기억이 나지 않는다고 머리를 쥐어 뜯지 마세요. 그냥 이전 강좌를 다시 보면 됩니다. 이전 강좌의 내용이 잘 기억이 나지 않는 것은 '지극히 정상' 이니 다시 한 번 읽어 보므로써 기억을 강화시키도록 하세요 :)

그럼 잡담을 끝내고 본론으로 들어가도록 합시다. 이전 강좌에서 배열은 변수 들의 모임이라고 했습니다. 이 때 `arr` 이라는 배열의 `i` 번째 원소를 참조 하기 위해선 `arr[i]` 라고 써야 한다는 것도 알았습니다. 그런데 똑똑한 사람이라면 이 아이디어를 확장해서 다음과 같은 생각을 할 수 도 있을 것 입니다.

> 배열의 배열을 만들면 어떨까?

정말로 놀라운 생각 입니다. 일단 여기서 우리는 **배열의 배열**의 의미를 좀 더 명확하게 해야 겠습니다. *int* 형의 배열 이란 말은 배열의 각 원소가 *int* 형 변수 인 것*!* 입니다.

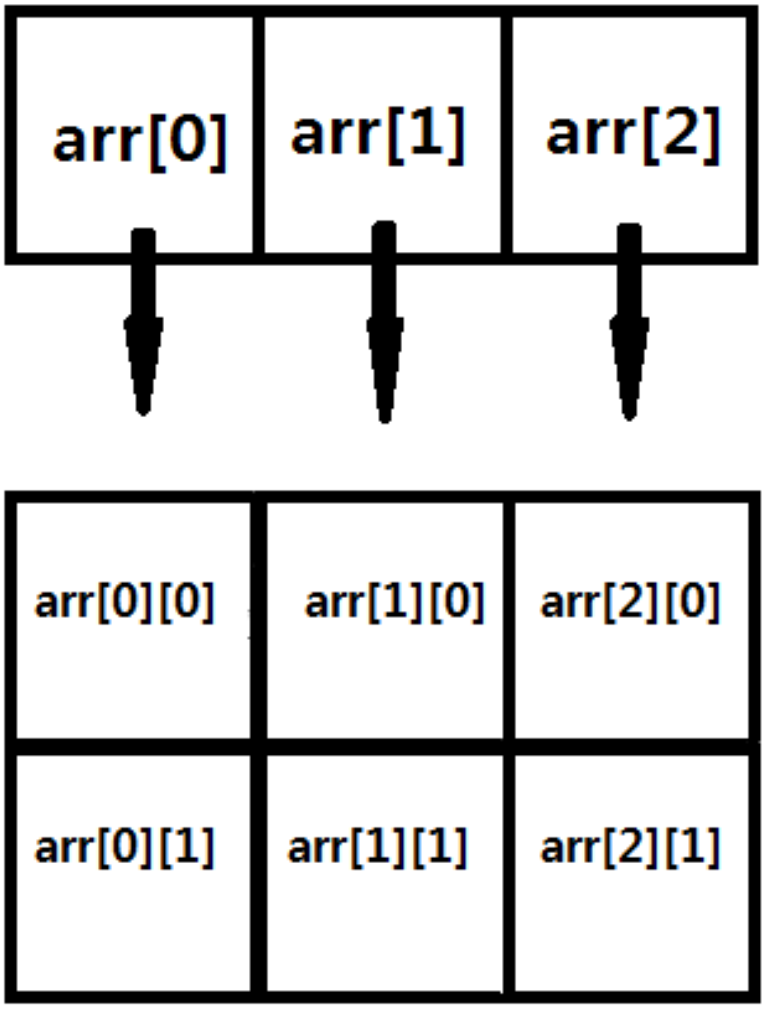
그렇다면 **int 형의 배열의 배열** 이란 말은 무엇일까요? 이 말은, **배열의 각 원소가 int 형의 배열인 것** 을 말합니다. C 언어 에서 이러한 형태의 배열을 정의하기 위해선 아래와 같이 하면 써주면 됩니다.

```
// 위 경우 (배열의 형) 부분에 int 가 오면 된다. ? 에는 배열의 크기
(배열의 형) (배열의 이름)[?][?];
```

이전에는 `(배열의 이름)[?]` 였지만 2 차원 배열은 옆에 `[?]` 하나가 더 붙었지요.먼저 배열의 배열이 무엇인지 확실하게 알기 위해서 다음과 같은 배열을 정의해 봅시다.

```
int arr[3][2];
```

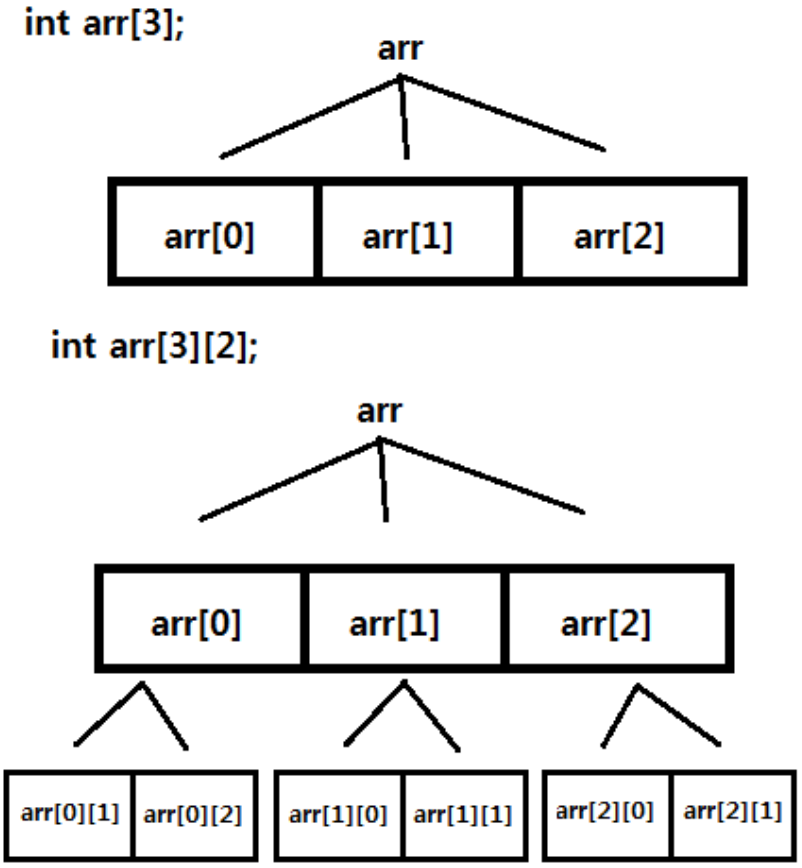
앞에서 말했듯이 위 배열은 '배열의 각 원소 3 개가 원소를 2 개 가지는 int 형의 배열이고 이름은 arr 이다.' 을 의미 합니다. 따라서,



가 됩니다.

즉, arr[0] 이라 하면 int 형의 원소를 2 개 가지는 배열을 말하는 것이며 그 배열의 원소는 각각 arr[0][0], arr[0][1] 이 되겠지요.

일차원 배열과 이차원 배열을 한 눈에 비교하자면 아래와 같습니다.



어때요, 간단하죠? 따라서, `arr[m][n];` 과 같이 배열을 선언한다면 (`m` 과 `n` 은 임의의 정수값), $m \times n$ 개의 변수를 가지는 배열을 선언한 것이 됩니다.

그렇다면, 2 차원 배열을 가지고 무슨 짓을 할 수 있을 까요? 사실, 여러분도 이미 예상한 바 있지만 오히려 일차원 배열에 비해 활용도가 훨씬 높아지게 됩니다. 예를 들면, 33 명의 학생에 대해 국어, 수학, 영어, 과학 점수를 보관하는 배열을 만든다 (이전의 배열 하나만 이용해선 한 과목의 점수 밖에 보관할 수 없었죠) 도서 입출 관리 프로그램에서 개개의 도서에 대해서 이 도서를 빌려간 날짜, 반납한 날짜 등을 보관하는 배열을 만든다 등등이 있습니다.

아! 그런데 아직 왜 이러한 배열을 '2차원' 배열이라고 말하는지 이야기 하지 않았군요. 사실, 메모리에는 모든 배열이 일차원 배열과 다름없이 들어갑니다. 그런데, 아래 예제를 보고 나면 왜 '2차원' 배열이라 이야기 하는지 감이 확 올 것 입니다.

```c
/* 2 차원 배열 */
#include <stdio.h>
int main() {
  int arr[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

  printf("arr 배열의 2 행 3 열의 수를 출력 : %d \n", arr[1][2]);
  printf("arr 배열의 1 행 2 열의 수를 출력 : %d \n", arr[0][1]);
  return 0;
```

```
}
```

성공적으로 컴파일 하였으면

실행 결과

```
arr 배열의 2 행 3 열의 수를 출력 : 6
arr 배열의 1 행 2 열의 수를 출력 : 2
```

와 같이 나옵니다. 처음에 2 차원 배열을 정의 할 때 부터 확 와닿는 느낌이 듭니다. 왜냐하면 정말로 2 차원 상에 배열된 배열이라고 생각할 수 있고, 배열의 선언 자체가 2 차원 적으로 정의 되었기 때문이죠

```
int arr[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

위에서 2 차원 배열을 정의 하였습니다. 그런데 사실 아래와 같이 모두 한 줄에

```
int arr[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

다 써도 큰 문제는 없지만 보기 좋기 위해 위와 같이 나열한 것 입니다.왜냐하면 2 차원 배열을 아래와 같은 모습으로 존재한다고 상상 할 수 있기 때문이죠.

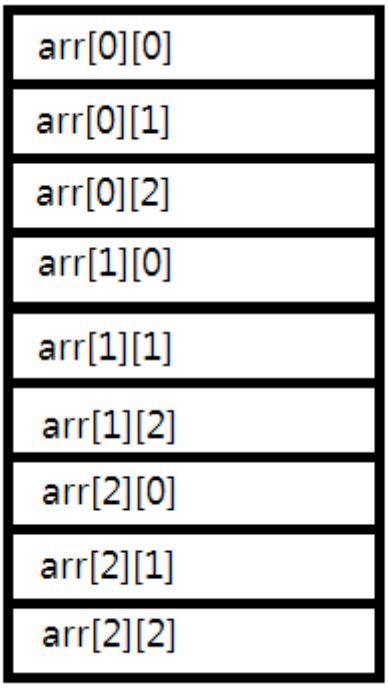
arr[0][0] 1	arr[0][1] 2	arr[0][2] 3
arr[1][0] 4	arr[1][1] 5	arr[1][2] 6
arr[2][0] 7	arr[2][1] 8	arr[2][2] 9

마치 우리가 배열을 정의했던 것 처럼 이차원 상에 배열이 있다고 생각해서 그려볼 수 있습니다. 이 때, `arr[x][y]` 라고 한다면 `x` 는 몇 번째 줄에 있는지, `y` 는 몇 번째 열에 있는지를 나타냅니다. 예를 들어서 `arr[0][1]` 은 0 번째 줄, 1 번째 열에 있으므로 2 가 되겠지요. 결과적으로 말하자면 '일차원 배열은 한 개의 값(x)으로 원소에 접근하는 것이고, 이차원 배열은 두 개의 값(x,y)으로원소에 접근하는 것이다!' 라고 생각할 수 있게 됩니다.

한가지 눈 여겨 볼 점은 arr 이 '배열의 배열' 이라는 것이 맞다는 것 입니다. 왜냐하면 arr[0] 을 하나의 배열의 이름이라고 생각해 보면 3 개의 원소 arr[0][0], arr[0][1], arr[0][2] 를 가진다고 볼 수 있다는 것 이지요. 마찬가지로 arr[1], arr[2] 도 하나의 배열이라고 생각할 수 있습니다. 그런데 arr 이 배열이므로 우리는 2 차원 배열을 '배열의 배열' 이라고 볼 수 있지요.

기존의 배열과 마찬가지로 arr[3] 이라 하면 arr[0] ~ arr[2] 까지 사용 가능했듯이 arr[3][3] 이라 하면 arr[0][0] ~ arr[0][2], arr[1][0] ~ arr[1][2], arr[2][0] ~ arr[2][2] 까지 사용 가능 합니다.

그런데 한 가지 지적해야 할 부분은 컴퓨터 메모리 상에선 절대로 이차원 적으로 만들어 지지 않는다는 것 입니다. 사실, 컴퓨터 메모리 상에선 2차원 이라는 것이 존재할 수 가 없습니다. 단지 선형으로 된 데이터들의 나열일 뿐이지요. 위 배열의 경우 컴퓨터 메모리 상에 다음과 같이 존재합니다.



하지만 우리가 이렇게 메모리 상에 선형으로 배열되어 있음에도 불구하고 '이차원 배열' 이라고 부르는 이유는 위 처럼 메모리 상에 2 차원으로 배열되어 있다고 생각하면 정말로 간편하기 때문입니다. 예를 들어서 arr[2][1] 은 메모리 상에서 배열의 시작 부분 (arr[0][0]) 에서 부터 3*2 + 1 = 7 번째에 있는 값 이라고 생각해야 되지만 사실 이 데이터를 2 차원 상에 배열해 놓고 2 행, 1 열의 값 이라고 생각하면 훨씬 편하기 때문입니다. (물론 컴퓨터는 전자의 경우로 계산하게 됩니다)

```c
/* 학생 점수 입력 받기 */
#include <stdio.h>
int main() {
  int score[3][2];
  int i, j;
```

```
  for (i = 0; i < 3; i++)  // 총 3 명의 학생의 데이터를 받는다
  {
    for (j = 0; j < 2; j++) {
      if (j == 0) {
        printf("%d 번째 학생의 국어 점수 : ", i + 1);
        scanf("%d", &score[i][j]);
      } else if (j == 1) {
        printf("%d 번째 학생의 수학 점수 : ", i + 1);
        scanf("%d", &score[i][j]);
      }
    }
  }

  for (i = 0; i < 3; i++) {
    printf("%d 번째 학생의 국어 점수 : %d, 수학 점수 : %d \n", i + 1,
           score[i][0], score[i][1]);
  }

  return 0;
}
```

성공적으로 컴파일 하였다면

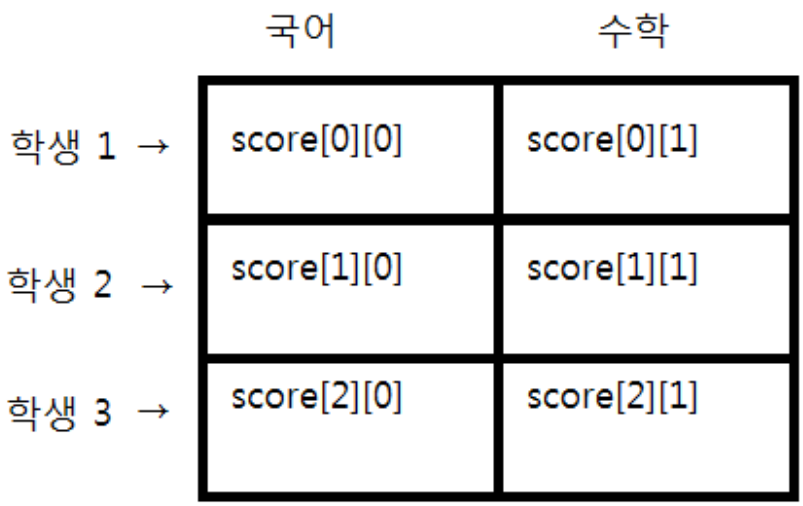
실행 결과

```
1 번째 학생의 국어 점수 : 30
1 번째 학생의 수학 점수 : 60
2 번째 학생의 국어 점수 : 90
2 번째 학생의 수학 점수 : 100
3 번째 학생의 국어 점수 : 70
3 번째 학생의 수학 점수 : 80
1 번째 학생의 국어 점수 : 30, 수학 점수 : 60
2 번째 학생의 국어 점수 : 90, 수학 점수 : 100
3 번째 학생의 국어 점수 : 70, 수학 점수 : 80
```

와 같이 됩니다. 사실 작동 원리는 간단 합니다.

```
int score[3][2];
```

일단 위 구문을 통해 3 행, 2 열의 크기를 가지는 2 차원 배열 score 을 선언 하였습니다. 사실 우리가 프로그래밍 하고자 하는 목표에 따라 해석해 보면 '3 명의 학생의 2 과목의 데이터를 보관하는 score 2 차원 배열' 이라고 볼 수 도 있습니다. 이를 그림으로 나타내면



꼴로 보면 됩니다.

```
for (i = 0; i < 3; i++)  // 총 3 명의 학생의 데이터를 받는다
{
  for (j = 0; j < 2; j++) {
    if (j == 0) {
      printf("%d 번째 학생의 국어 점수 : ", i + 1);
      scanf("%d", &score[i][j]);
    } else if (j == 1) {
      printf("%d 번째 학생의 수학 점수 : ", i + 1);
      scanf("%d", &score[i][j]);
    }
  }
}
```

이제 for 문을 통해서 3 명의 학생의 데이터를 입력 받게 됩니다. 일단 오래간만에 두 개의 for 문이 같이 돌아가는데 어떠한 형식으로 작동되는 지는 알고 있겠지요? i = 0 일 때, j = 0 ~ 1, i = 1 일 때, j = 0 ~ 1, i = 2 일 때, j = 0 ~ 1 로 돌아가게 됩니다. 즉 위 부분을 통해 2 차원 score 배열의 값을 집어 넣게 되는 것 이지요.

```
if (j == 0) {
  printf("%d 번째 학생의 국어 점수 : ", i + 1);
  scanf("%d", &score[i][j]);
} else if (j == 1) {
  printf("%d 번째 학생의 수학 점수 : ", i + 1);
```

```
  scanf("%d", &score[i][j]);
}
```

위 for 문 안의 위 부분을 살펴 보면 j 가 0 이면 국어점수를 입력해라, j 가 1 이면 수학 점수를 입력해라 라고 물어 보는 것이 달라 집니다.

마지막으로

```
for (i = 0; i < 3; i++) {
  printf("%d 번째 학생의 국어 점수 : %d, 수학 점수 : %d \n", i + 1, score[i][0],
         score[i][1]);
}
```

를 통해 입력 받은 값을 깔끔하게 보여주게 됩니다.

2 차원 배열 정의하기

앞선 예제에서

```
int arr[2][3] = {1, 2, 3, 4, 5, 6};
```

와 같이 2 차원 배열을 선언하였습니다. 그런데, 프로그래밍시 줄 수를 절약하고 싶은 사람들은 아래와 같이 해도 큰 문제는 없습니다.

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

위 처럼 정의하는 것은 우리가 앞서 써 왔던 방법과 전혀 차이가 없으니 그냥 알아 두시면 됩니다. 여러분들 께서 원하는 방법으로 정의하시면 됩니다.

참고로, 이전 강의에서 이야기를 하지 않았는데, 다음과 같이 배열을 정의할 수 도 있습니다.

```
int arr[] = {1, 2, 3, 4};
```

음... 무언가 이상하다는 느낌이 드나요? 사실, 알고보면 단순합니다. 위와 같이 정의할 수 있는 이유는 컴파일러가 원소의 개수를 정확하게 알기 때문입니다. 컴파일러는 우리가 배열을 정의한 것을 보고 '아, 이 사람이 원소를 4 개 가지는 int 배열을 정의하였구나!' 라고 알아서 대괄호 안에 자동적으로 4 를 집어 넣어서 생각하게 됩니다. 따라서 위와 같이 정의하나

```
int arr[4] = {1, 2, 3, 4};
```

로 정의하나 같은 말이 되겠지요.하지만 아래와 같이 정의하는 것은 안됩니다.

```
int arr[];
```

이 것은 왜 안될까요? 아마, 이전 강의를 잘 보신 분들게서는 단박에 알아 차릴 수 있을 것 입니다. 그 이유는 '배열의 크기는 임의로 정해지지 않기 때문입니다. 즉, 위와 같이 배열을 정의한다면 컴파일러는 우리가 어떠한 크기의 배열을 정의하고 싶은지 모릅니다. 따라서 아래와 같은 오류를 내뿜게 됩니다.

컴파일 오류

```
error C2133: 'arr' : 알 수 없는 크기입니다.
```

이 아이디어를 2 차원 배열에도 그대로 적용 시킬 수 있습니다.일단, 아래와 같은 2 차원 배열의 정의를 살펴 보도록 합시다.

```
int arr[][3] = {{4, 5, 6}, {7, 8, 9}};
```

위 정의를 보면 여러분은 비어있는 대괄호 안에 무슨 값이 들어갈지 맞출 수 있을 것입니다. 무엇이냐고요? 바로 2 이지요. 왜냐하면 {4,5,6} 를 가지는 arr[3] 배열 하나와, {7,8,9} 를 가지는 또다른 arr[3] 배열을 정의하여 총 2 개의 arr[3] 배열을 정의하였기에, int arr[2][3] 이 되어야 하는 것이지요.

그렇다면 아래와 같은 문장이 유효한지 살펴보세요.

```
int arr[][2] = {{1, 2}, {3, 4}, {5, 6}, {7}};
```

어! 이상하네요. 마지막에 그냥 {7} 이라고 되어 있잖아요? 아마 여러분들 중 대다수는 이 것을 보고 위 문장이 틀렸고 성공적으로 컴파일 되지 않으리라 생각할 것입니다. 하지만, 위 2 차원 배열은 배열 정의시 arr[][2] 라고 하였기 때문에 무조건 원소가 2 인 1 차원 배열들이 생기게 됩니다. 즉, 7 이 속한 1 차원 배열에는 원소가 한 개인 것이 아니라 마치 arr[3] = {1} 고 해도 상관없는 것 처럼 8 번째 원소가 들어갈 자리를 비워놓게 됩니다. 따라서, 위 문장은 틀린 것이 아닙니다. 그렇다면 아래 문장을 봐보세요.

```
int arr[2][] = {{4, 5, 6}, {7, 8, 9}};
```

과연 될까요? 아마 여러분들 중 대다수는 될 것이라 생각하고 있을 것 입니다. 하지만 놀랍게도 컴파일해보면

컴파일 오류

```
error C2087: 'arr' : 첨자가 없습니다.
error C2078: 이니셜라이저가 너무 많습니다.
```

와 같은 오류들을 만나게 됩니다. **C 에서는 다차원 배열의 경우 맨 앞의 크기를 제외한 나머지 크기들을 정확히 지정해줘야 오류가 발생하지 않습니다.**

3 차원, 그 이후 차원의 배열들

2 차원 배열을 잘 이해하였다면 3 차원 배열을 이해하는 것은 그리 어려운 것이 아니라 생각됩니다. 사실, 보통의 프로그래밍에서 3 차원 배열을 쓰는 경우는 그렇게 많지 않습니다. (물론 제가 만들어본 프로그램들에 한해서...) 그렇지만 쓸 수 도 있기에 간단하게 집고 넘어가기만 합시다.

3 차원의 배열의 정의는 2 차원 배열과 거의 동일합니다. (그 이후의 차원들도 마찬가지)

```
(배열의 형)(배열의 이름)[x][y][z];  // 여기서 x,y,z 는 배열의 크기를 말합니다.
```

이제, 머리속으로 상상의 나래를 펼쳐 봅시다. 제가 그림판으로 3 차원 적인 그림을 그릴 수는 없으므로 여러분의 지능을 믿겠습니다! 일단, 아래의 배열을 머리에 그려 봅시다.

```
int arr[3][4];
```

이는 가로 길이가 4 이고 세로 길이가 3 인 평면위에 int 변수들이 하나씩 놀고 있는 것을 상상하면 됩니다. 그렇다면 이제 아래 배열을 머리에 그려 봅시다.

```
int brr[2][3][4];
```

아아악! 모르겠다고요? 아니요, 어렵지 않습니다. 위에서 상상한 평면 위에 동일한 평면이 한 층 더 있다고 생각하면 됩니다. 즉, 위에서 생각했던 평면이 2 개의 층으로 생겼다고 하면 됩니다. 어때요, 간단하죠?

하지만, 문제는 4차원 배열 부터 입니다. 뭐 우리는 3 차원 적인 세상에서 살고 있기 때문에 4 차원에 무엇인지 몸에 와닿기는 힘듭니다. (사실, 우리는 3차원 상의 공간에 시간의 축이 더해진 4차원 세상에서 살고 있다고 합니다) 그렇기에 4 차원 배열, 그리고 그 보다 더 놓은 차원의 배열이 무엇인지 머리속으로 그려보기란 고역이 아닐 수 없습니다.

그런데 말이죠. 제가 아까 굵은 글씨로 써 놓았던 것이 기억나시나요?

일차원 배열은 한 개의 값(x)으로 원소에 접근하는 것이고, 이차원 배열은 두 개의 값(x,y)으로 원소에 접근하는 것이다!

이를 확장해서 생각해 보면 삼차원 배열은 세 개의 값 `(x,y,z)` 을 통해서 원소에 접근하는 것입니다. 네, 맞아요. 우리가 원소가 몇 번째 층에 있고 (x), 그 층에 해당하는 평면에 몇 행(y), 그리고 몇 열(z) 를 알면 `int` 변수에 정확하게 접근할 수 있지 않습니까?

4 차원도 같습니다. 4 차원 배열은 4 개의 값 `(x,y,z,w)` 을 통해서 원소에 접근할 수 있습니다. 마찬가지로 5 차원은 5 개, `n` 차원은 `n` 개의 값을 통해서 원소에 접근하게 되는 것이지요. 이 아이디어를 적용시키면 어떠한 차원의 배열이 실제 프로그래밍 상에 필요하다고 하더라도 문제 없이 해결할 수 있으리라 생각합니다.

그렇다면 이번 강좌는 여기에서 마치도록 하겠습니다.

생각해 보기

문제 1

제 강좌 제목에서 배열이 왜 C 언어의 아파트 인지 설명해 보세요. 즉, 동 의 개념, 층 의 개념, 호 의 개념이 어떠한 배열을 형상화 하고 있는 지도 생각해 보세요. (난이도 : 下)

10

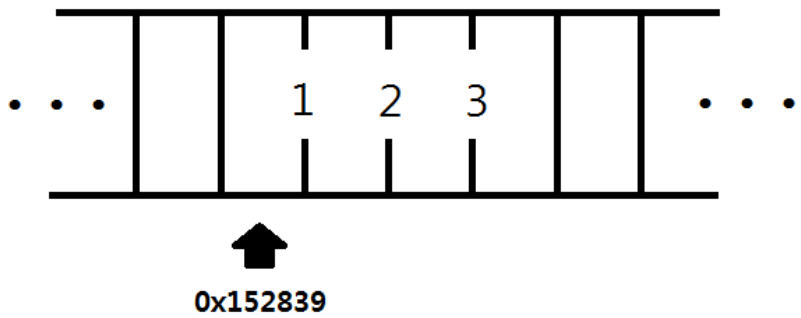
포인터

우왕~ 안녕하세요 여러분. 아마 C 언어를 배웠거나 배우고 있는 사람들은 포인터에 대해 익히 들어보셨을 것 입니다. 이해하기 힘들기로 악명 높은 그 포인터를 말이죠. 하지만, 저와 함께 한다면 큰 무리 없이 배우 실 수 있을 것이라 생각됩니다.

포인터를 이해하기 앞서

앞서 3 강에서 이야기 하였지만 모든 데이터들은 메모리 상에 특정한 공간에 저장 되어 있습니다. 우리는 앞으로 편의를 위해, 메모리의 특정한 공간을 '방' 이라고 하겠습니다. 즉, 각 방에는 데이터들이 들어가게 되는 것 입니다. 보통 사람들은 한 방의 크기를 1 바이트 라고 생각합니다. 우리가 만약 `int` 형 변수를 정의한다면 4 바이트 이므로 메모리 상의 4 칸을 차지하게 됩니다. 그런데 말이죠. 프로그램 작동 시 컴퓨터는 각 방에 있는 데이터를 필요로 하게 됩니다. 따라서, 서로 구분하기 위해 각 방에 고유의 **주소(address)** 를 붙여 주었습니다. 우리가 아파트에서 각 집들을 호수로 구분하는 것 처럼 말입니다. 예를 들어 우리가 아래와 같은 `int` 변수 `a` 를 정의하였다면 특정한 방에 아래 그림 처럼 변수 `a` 가 정의됩니다.

```
int a = 123;  // 메모리 4 칸을 차지하게 한다.
```



이 때, `0x152839` 는 제가 아무렇게나 정한 이 방의 시작 주소 입니다. 참고로, `0x` 가 뭐냐고 물어보는 사람들이 있을 텐데, 이전 강좌에서도 이야기 하였지만 16 진수라고 표시한 것 입니다. 즉, 16 진수로 `152839` (10 진수로 1386553) 라는 위치에서 부터 4 바이트의 공간을 차지하며 123 이라는 값이 저장되어 있게 하라는 뜻이지요.

그렇다면 아래와 같은 문장은 어떻게 수행 될까요?

```
a = 10;
```

사실 컴파일러는 위 문장을 아래와 같이 바꿔주게 됩니다.

메모리 `0x152839` 위치에서 부터 4 바이트의 공간에 있는 데이터를 10 으로 바꾸어라!

결과적으로, 컴퓨터 내부에서는 올바르게 수행되겠지요.

참고적으로 말하는 이야기 이지만 현재 (아마 이 블로그에 접속하는 사람들 중 99% 이상이) 많은 사람들은 32 비트 운영체제를 사용하고 있습니다.[1] 이 32 비트에서 작동되는 컴퓨터들은 모두 주소값의 크기가 32 비트 (즉, 4 바이트.. 까먹었다면 2 - 3 강 참조) 로 나타내집니다. 즉 주소값이 `0x00000000` ~ `0xFFFFFFFF` 까지의 값을 가진다는 것이지요.

어랏! 조금 똑똑하신 분들이라면 32 비트로 사용할 수 있는 주소값의 가지수는 2 의 32 승 바이트, 즉 RAM 은 최대 4 GB 까지 밖에 사용할 수 없다는 사실을 알 수 있습니다. 맞습니다. 이 때문에 32 비트 운영체제에서는 RAM 의 최대 크기가 4 GB 로 제한되지요(즉, 돈을 많이 들여서 RAM 을 10GB 로 만들어도 컴퓨터는 4 GB 까지 밖에 인식하지 못합니다. 어찌 이렇게 슬플수가..)

여기까지는 상당히 직관적이고 단순해서 이해하기 쉬웠을 것 입니다. 그런데 C 를 만든 사람은 아주 유용하면서도 골때리는 것을 하나 새롭게 만들었습니다. 바로 '포인터(pointer)' 입니다. 영어를 잘하는 분들은 이미 아시겠지만 '포인터' 라는 단어의 뜻이 '가리키는 것(가르켜지는 대상체를 말하는 것이 아닙니다)' 이란 의미를 가지고 있습니다.

사실, 포인터는 우리가 앞에서 보았던 `int` 나 `char` 변수들과 다른 것이 전혀 아닙니다. 포인터도 '변수' 입니다. `int` 형 변수가 정수 데이터, `float` 형 변수가 실수 데이터를 보관했던 것 처럼, 포인터도 특정한 데이터를 보관하는 '변수' 입니다. 그렇다면 포인터는 무엇을 보관하고 있을 까요?

바로, 특정한 데이터가 저장된 주소값을 보관하는 변수 입니다. 여기서 강조할 부분은 '주소값' 이라는 것 이지요. 여기서 그냥 머리에 박아 넣어 버립시다. 이전에 다른 책들에서 배운 내용을 싹 다 잊어 버리고 그냥 망치로 때려 넣듯이 박아버려요. 포인터에는 특정한 데이터가 저장된 주소값을 보관하는 변수 라고 말이지요. 크게 외치세요. '주소값!!!!!'

암튼, 뇌가 완전히 세뇌되었다고 생각하면 다음 단계로 넘어가도록 하겠습니다. 아직도 이상한 잡념이 머리에 남아 있다면 크게 숨을 호흡하시고 주소값이라고 10 번만 외쳐 보세요..

자. 되었습니다. 이제 포인터의 세계로 출발해 봅시다. 뿡

1) 참고로 강좌가 처음 제작된 년도가 2009년 입니다. 지금 (2018년 이후) 이 글을 보시는 분들은 아마 대부분 64 비트 운영체제를 사용하고 있을 것입니다.

포인터

다시 한 번 정리하자면

> 포인터 : 메모리 상에 위치한 특정한 데이터의 (시작)주소값을 보관하는 변수

우리가 변수를 정의할 때 `int` 나 `char` 처럼 여러가지 **형(type)** 들이 있었습니다. 그런데 놀랍게도 포인터에서도 형이 있습니다.

이 말은 포인터가 메모리 상의 `int` 형 데이타의 주소값을 저장하는 포인터와, `char` 형 데이터의 주소값을 저장하는 포인터가 서로 다르다는 말입니다. 응?? 여러분의 머리속에는 아래와 같은 생각이 번개 처럼 스쳐 지나갈 것입니다.

> 아까 포인터는 주소값을 저장하는 거래며. 근데 우리가 쓰는 컴퓨터에선 주소값이 무조건 32 비트, 즉 4 바이트래며! 그러면 포인터의 크기는 다 똑같은것 아냐? 근데 왜 포인터가 형(type)을 가지는 건데?!

휴우우. 진정좀 하시고. 여러분 말이 백번 맞습니다 - 단, 현재 까지 배운 내용을 가지고 생각하자면 말이지요. 포인터를 아주 조금만 배우면 왜 포인터에 형(type) 이 필요한지 알게 될 것입니다.

C 언어에서 포인터는 다음과 같이 정의할 수 있습니다 C 언어에서 포인터는 다음과 같이 정의할 수 있습니다.

> (포인터에 주소값이 저장되는 데이터의 형) *(포인터의 이름);

혹은 아래와 같이 정의할 수 도 있습니다.

> (포인터에 주소값이 저장되는 데이터의 형)* (포인터의 이름);

예를 들어 `p` 라는 포인터가 `int` 데이터를 가리키고 싶다고 하면

```
int* p;  // 라고 하거나
int* p;  // 로 하면 된다
```

라 하면 올바르게 됩니다. 즉 위 포인터 `p` 는 `int` 형 데이터의 주소값을 저장하는 변수가 되는 것입니다. 와우!

& 연산자

그런데 말입니다. 아직도 2% 부족합니다. 포인터를 정의하였으면 값을 집어 넣어야 하는데, 도대체 우리가 데이터의 주소값을 어떻게 아냐는 말입니까? 걱정 마십시요. 바로 & 연산자를 사용하면 됩니다.

그런데, 아마 복습을 철저하게 잘하신 분들은 당황할 수 도 있습니다. 왜냐하면 & 가 AND 연산자이기 때문입니다. (4 강 참조) 그런데, & 연산자를 사용하기 위해서는 두 개의 피연산자를 사용해야 합니다. 즉,

```
a& b;  // 괜찮음
a&     // 오류
```

와 같이 언제나 2 개가 필요 하다는 것이지요. 그런데, 여기에서 소개할 & 연산자는 오직 피연산자가 1 개인 연산자 입니다. (이러한 연산자를 **단항(unary)** 연산자라 합니다) 따라서 위의 AND 연산자와 완전히 다르게 해석됩니다.

단항 & 연산자는 피연산자의 주소값을 불러 옵니다. 사용하는 방법은 그냥

```
&/* 주소값을 계산할 데이터 */
```

예를 들어서 어떤 변수 a 의 주소값을 알고 싶다면

```
&a
```

로 쓰면 됩니다!

백설(說)이 불여일행(行). 한 번 프로그램을 짜 봅시다.

```
/* & 연산자 */
#include <stdio.h>
int main() {
  int a;
  a = 2;

  printf("%p \n", &a);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
0x7fff80505b64
```

와 같이 나옵니다. 참고로, 여러분의 컴퓨터에 따라 결과가 다르게 나올 수 도 있습니다. 사실, 저와 정말 인연 이상의 무언가가 있지 않는 이상 전혀 다르게 나올 것 입니다. 더 놀라운 것은 실행할 때 마다 결과가 달라질 것입니다.

2 번째 실행한 것

실행 결과

```
0x7ffe37d03104
```

위와 같이 나오는 이유는 나중에 설명하겠지만 주목할 것은 어떠한 **값**이 출력되었다는 것 입니다.

```
printf("%x \n", &a);
```

위 문장에서 &a 의 값을 16 진수 형태 (%p) 로 출력하라고 명령하였습니다. 근데요. 눈치가 있는 사람이라면 금방 알겠지만 위에서 출력된 결과는 8 바이트 (16 진수로 16 자리)가 아닙니다! (여러분의 컴퓨터는 다를 수 있습니다.) 제가 지금 64 비트 운영체제를 사용하고 있는데도 말이지요!

그렇다면 뭐가 문제인가요? 사실, 문제는 없습니다. 단순히 앞의 0 이 잘린 것 이지요. 주소값은 언제나 8 바이트 크기, 즉 16 진수로 16 자리 인데 앞에 0 이 잘려서 출력이 안된 것일 뿐입니다. 따라서 변수 a 의 주소는 아마도 `0x000007ffe37d03104` 가 될 것입니다.[2)]

아무튼 위 결과를 보면, 적어도 제 컴퓨터 상에선 `int` 변수 a 는 메모리 상에서 `0x7ffe37d03104` 를 시작으로 4 바이트의 공간을 차지하고 있었다는 사실을 알 수 있습니다.

자, 이제 & 연산자를 사용하여 특정한 데이터의 메모리 상의 주소값을 알 수 있다는 사실을 알았으니 배고픈 포인터에게 값을 넣어 봅시다.

```
/* 포인터의 시작 */
#include <stdio.h>
int main() {
  int *p;
  int a;

  p = &a;

  printf("포인터 p 에 들어 있는 값 : %p \n", p);
```

2) 아마 지금(2018년 이후) 강좌를 보고 계시는 분들은 앞에서도 말했듯이 64 비트 시스템을 사용하고 계실 가능성이 높기 때문에, 4 바이트가 아니라 8 바이트 숫자가 출력될 것입니다.

```
  printf("int 변수 a 가 저장된 주소 : %p \n", &a);

  return 0;
}
```

실행해 보면 많은 이들이 예상했던 것 처럼....

실행 결과

```
포인터 p 에 들어 있는 값 : 0x7fff894c8b3c
int 변수 a 가 저장된 주소 : 0x7fff894c8b3c
```

똑같이 나옵니다. 어찌 보면 당연한 일입니다.

```
p = &a;
```

에서 포인터 p 에 a 의 주소를 대입하였기 때문이죠. 참고로, 한 번 정의된 변수의 주소값은 바뀌지 않습니다. 따라서 아래 printf 에서 포인터 p 에 저장된 값과 변수 a 의 주소값이 동일하게 나오게 됩니다. 어때요. 쉽죠?

* 연산자

현재 까지 우리가 배운 바로는 **포인터는 특정한 데이터의 주소값을 보관한다. 이 때 포인터는 주소값을 보관하는 데이터의 형에 * 를 붙임으로써 정의되고, & 연산자로 특정한 데이터의 메모리 상의 주소값을 알아올 수 있다** 였습니다.

& 연산자가 어떠한 데이터의 주소값을 얻어내는 연산자라면 거꾸로 주소값에서 해당 주소값에 대응되는 데이터를 가져오는 연산자가 필요하겠지요? 이 역할은 바로 * 연산자가 수행합니다!

잠깐만. * 연산자는 이미 곱셈 연산자로 사용되고 있지 않나요? 맞습니다. 다만, * 연산자가 피연산자 두 개에 작용할 때만 곱셈 연산자로 해석됩니다. 즉,

```
a * b; // a 와 b 를 곱한다.
a *; // 오류!
*a; // 단항 * 연산자
```

와 같이 해석됩니다.

* 연산자의 역할을 쉽게 풀이하자면

> "나(포인터)를 나에게 저장된 주소값에 위치한 데이터로 생각해줘!"

의 역할을 수행합니다. 한 번 아래 예제를 봅시다.

```c
/* * 연산자의 이용 */
#include <stdio.h>
int main() {
  int *p;
  int a;

  p = &a;
  a = 2;

  printf("a 의 값 : %d \n", a);
  printf("*p 의 값 : %d \n", *p);

  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
a 의 값 : 2
*p 의 값 : 2
```

가 됩니다.

```c
int *p;
int a;
```

일단 int 데이터를 가리키는 포인터 p 와 int 변수 a 를 각각 정의하였습니다. 평범한 문장 이지요.

```c
p = &a;
a = 2;
```

그리고 포인터 p 에 a 의 주소를 집어 넣었습니다. 그리고 a 에 2 를 대입하였습니다.

```c
printf("a 의 값 : %d \n", a);
printf("*p 의 값 : %d \n", *p);
```

일단 위의 문장은 단순 합니다. a 의 값을 출력하란 말이지요. 당연하게도 2 가 출력됩니다. 그런데, 아래에서 *p 의 값을 출력하라고 했습니다. * 의 의미는 앞서, 나에 저장된 주소값에 해당하는 데이터로 생각하시오! 로 하게 하는 연산자라고 하였습니다.

따라서 *p 를 통해 p 에 저장된 주소(변수 a 의 주소)에 해당하는 데이터, 즉 변수 a 그 자체를 의미할 수 있게 됩니다.

다시 말해 *p 와 변수 a 는 정확히 동일합니다. 즉, 위 두 문장은 아래 두 문장과 백프로 일치합니다.

```
printf("a 의 값 : %d \n", a);
printf("*p 의 값 : %d \n", a);
```

마지막으로 * 와 관련된 예제 하나를 더 살펴 봅시다.

```
/* * 연산자 */
#include <stdio.h>
int main() {
  int *p;
  int a;

  p = &a;
  *p = 3;

  printf("a 의 값 : %d \n", a);
  printf("*p 의 값 : %d \n", *p);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

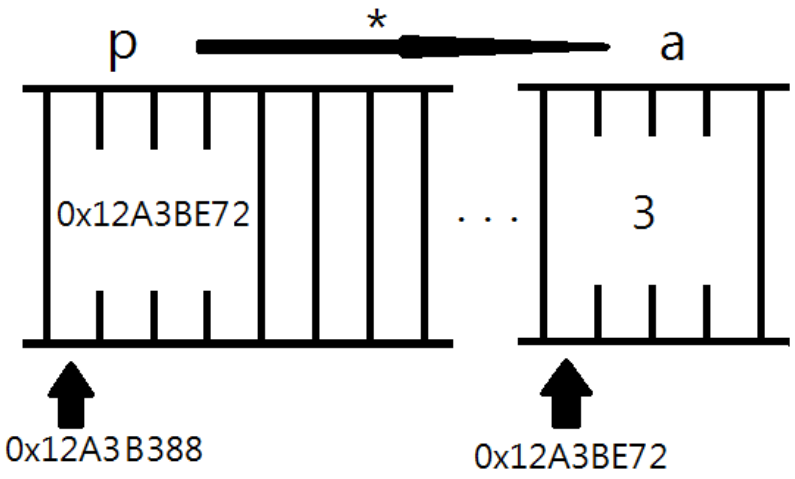
```
a 의 값 : 3
*p 의 값 : 3
```

아마 많은 여러분들이 예상했던 결과 이길 바랍니다!

```
p = &a;
*p = 3;
```

위에서도 마찬가지로 p 에 변수 a 의 주소를 집어 넣었습니다. 그리고 *p 를 통해 "나에 저장된 주소(변수 a 의 주소)에 해당하는 데이터(변수 a) 로 생각하시오" 를 의미하여 *p = 3 은 a = 3 과 동일한 의미를 지니게 되었습니다. 어때요. 간단하지요? 이로써 여러분은 포인터의 50% 이상을 이해하신 것 입니다~! 짝짝짝짝

자. 그럼 **포인터** 라는 말 자체의 의미를 생각해 봅시다. int 변수 a 와 포인터 p 의 메모리 상의 모습을 그리면 아래와 같습니다.



참고로 주소값은 제가 임의로 정한 것 입니다.

포인터 p 에 어떤 변수 a 의 주소값이 저장되어 있다면 **포인터 p 는 변수 a 를 가리킨다** 라고 말합니다. 포인터 또한 엄연한 변수 이기 때문에 특정한 메모리 공간을 차지합니다. 따라서 위 그림과 같이 포인터도 자기 자신만의 주소를 가지고 있습니다.

포인터에는 왜 타입이 있을까

여기 까지 왔다면 아마 다음과 같은 의문을 가질 수 있을 것입니다.

> 포인터가 주소값만 보관하는데 왜 굳이 타입이 필요할까? 어차피 주소값은 32 비트 시스템에서 항상 4 바이트이고, 64 비트 시스템에서는 8 바이트 인데 그냥 pointer 라는 타입을 만들어버리면 안될까?

아주 좋은 질문 입니다. pointer 라는 타입이 있다고 생각해고 아래의 코드를 살펴봅시다.

```
int a;
pointer *p;
p = &a;
*p = 4;
```

컴퓨터 입장에서 위 코드를 어떤 식으로 해석할 지 생각해볼까요.

```
int a;
pointer *p;
p = &a;
```

위 세 문장 까지는 아주 좋습니다. 메모리에 a 를 위해서 4 바이트 짜리 공간을 마련해줬고, 마찬가지로 p 를 위해 메모리 상에 8 바이트 짜리 공간을 마련하였습니다. 그리고 p 에 a 의 주소값을 잘 전달하였지요.

문제는 아래 문장 입니다.

```
*p = 4;
```

포인터 p 에는 명백히 변수 a 의 주소값이 들어 있습니다. 여기서 문제는 a 가 메모리에서 차지하는 모든 주소들의 위치가 들어 있는 것이 아니라 **시작 주소** 만 들어가 있다는 점입니다.

따라서, *p 라고 했을 때 컴퓨터는 메모리에서 얼마만큼을 읽어들어야 할지 알 길이 없습니다.

한편

```
int a;
int *p;
p = &a;
*p = 4;
```

라고 한다면 어떨 까요? 컴퓨터는 포인터 p 가 `int *` 라는 사실을 보고 **이 포인터는 `int` 데이터를 가리키는 구나!** 라고 알게 되어 시작 주소로 부터 정확히 4 바이트를 읽어 들어 값을 바꾸게 됩니다.

포인터도 변수다

```
/* 포인터도 변수이다 */
#include <stdio.h>
int main() {
  int a;
  int b;
  int *p;

  p = &a;
  *p = 2;
  p = &b;
  *p = 4;

  printf("a : %d \n", a);
  printf("b : %d \n", b);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
a : 2
b : 4
```

```
p = &a;
*p = 2;
p = &b;
*p = 4;
```

사실, 이런 예제까지 굳이 보여주어야 하나 하는 생각이 들었지만 그래도 혹시나 하는 마음에 했습니다. 앞에서도 말했듯이 포인터는 **변수** 입니다.

즉, 포인터에 들어간 주소값이 바뀔 수 있다는 것이지요. 위와 같이 처음에 a 를 가리켰다가, (즉 p 에 변수 a 의 주소값이 들어갔다가) 나중에 b 를 가리킬 수 (즉 p 에 변수 b 의 주소값이 들어감) 있다는 것 이지요. 뭐 특별히 중요한 예제는 아니였습니다만. 나중에 상수 포인터, 포인터 상수에 대해 이야기 하면서 다시 다루어 보도록 하겠습니다.

마지막으로, 강의를 마치며 여러분에게 포인터에 대해 완벽히 뇌리에 꽂힐 만한 동화를 들려드리겠습니다.

옛날 옛날에 대략 2 년 전에 (뭐.. 전 여러분과 옛날의 정의가 다릅니다ㅋ) 변철수, 변수철, 포영희라는 세 명의 사람이 OO 아파트에 살고 있었습니다.

```
int chul, sue;
int *young;
```

그런데 말이죠. 포영희는 변철수를 너무나 좋아한 나머지 자기 집 대문 앞에 큰 글씨로 "우리집에 오는 것들은 모두 철수네 주세요" 라고 써 놓고 철수네 주소를 적어 놓았습니다

```
young = &chul;
```

어느날 택배 아저씨가 영희네 집에 물건을 배달하러 왔다가 영희의 메세지를 보고 철수네에 가져다 주게 됩니다.

```
*young = 3; // 사실 chul = 3 과 동일하다!
```

영희에 짝사랑이 계속 되다가 어느날 영희는 철수 보다 더 미남인 수철이를 보게 됩니다. 결국 영희는 마음이 변심하고 수철이를 좋아하기로 했죠. 영희는 자기 대문 앞에 있던 메세지를 떼 버리고 "우리집에 오는 것은 모두 수철이네 주세요." 라 쓰고 수철이네 주소를 적었습니다.

```
young = &sue;
```

며칠이 지나 택배 아저씨는 물건을 배달하러 영희네에 왔다가 메세지를 보고 이번엔 수철이네에 가져다 줍니다.

```
*young = 4; // 사실 sue = 4 와 동일하다
```

이렇게 순수한 사랑이 OO 아파트에서 모락 모락 피어났습니다..... 끝

```
return 0; // 종료를 나타내는 것인데, 아직 몰라도 되요. (정확히 말하면 리턴...)
```

생각해 볼 문제

문제 1

`*` 와 `&` 연산자의 역할이 무엇인지 말해보세요 (난이도 : 下)

문제 2

`int **a;` 와 같은 이중 포인터(double-pointer) 에 대해 생각해 보세요 (난이도 : 中上)

상수 포인터, 포인터의 덧셈 뺄셈, 배열과 포인터

안녕하세요 여러분! 지난 시간에 포인터의 기본 중의 기본이라 할 수 있는 것 들에 배워보았습니다. 다시 정리해 보자면 포인터는 특정한 데이터의 메모리 상의 (시작) 주소값을 보관하는 변수 입니다.

제가 C 언어를 배우면서 포인터를 배울 때 가장 많이 든 생각은

> 근데 말야. 이거왜 배워?

이였습니다. 맞아요. 여러분들도 위와 같은 생각이 머리속에 끊임없이 맴돌 것 입니다. `int a;` 와 `int *p;` 가 있을 때 `p` 가 `a` 를 가리킨다고 하면 `a = 3;` 이라 하지 `*p = 3;` 과 같이 귀찮게 할 필요가 없잖아요. 하지만 나중에 가면 알겠지만 포인터는 C 언어에서 정말로 중요한 역할을 담당하게 될 것입니다. 포인터의 중요한 역할에 대해 지금 이야기 하는 것은 무리라고 생각합니다. 일단, 포인터가 뭔지만 알아 놓고 이걸 도대체 왜 배우는지에 대해선 나중에 이야기 하도록 합시다.

상수 포인터

이전에 11 - 1 강에서 상수에 대해 잠깐 언급한 것이 기억이 나시나요? 그 때 저는 어떠한 데이터를 상수로 만들기 위해 그 앞에 `const` 키워드를 붙여주면 된다고 했습니다. 예를 들어서

```
const int a = 3;
```

과 같이 값이 3 인 `int` 변수 `a` 를 상수로 정의할 수 있습니다. `const` 는 단순히 말해서 '이 데이터의 내용은 절대로 바뀔 수 없다' 라는 의미의 키워드 입니다. 따라서, 위 문장의 의미는 '이 `int` 변수 `a` 의 값은 절대로 바뀌면 안된다!!!' 가 됩니다. 위와 같이 정의한 상수 `a` 를 아래 문장에

```
a = 4;
```

와 같이 하려고 해도 컴파일 시에 오류가 발생하게 됩니다. 왜냐하면 `a` 는 상수로 선언이 되어 있으므로 값이 절대로 변경될 수 없기 때문이죠. 심지어 '값이 변경될 가능성이 있는 문장' 조차 허용되지 않습니다. 예를 들어

```
a = 3;
```

이라고 한다면, a 의 값은 이미 3 이므로 a 의 값은 바뀌지 않습니다. 그런데 왠일? 컴파일 해보면 오류가 출력됩니다. 왜냐하면 위 문장은 a 의 값이 바뀔 '가능성' 이 있기 때문이죠. 즉, 컴파일러는 a 에 무슨 값이 들어가 있는지 신경 쓰지 않습니다. 그냥 무조건 가능성이 있다면 오류를 출력합니다.

여러분은 도대체 왜 상수를 사용하는지 의문을 가질 것 입니다. 하지만 상수는 프로그래밍 상에서 프로그래머들의 실수를 줄여주고, 실수를 했다고 해도 실수를 잡아내는데 중요한 역할을 하고 있습니다. 예를 들어 아래와 같은 문장을 봅시다.

```
const double PI = 3.141592;
```

즉 double 형 변수 PI 를 3.141592 라는 값을 가지게 선언하였습니다. 왜 이렇게 해도 되냐면 실제로 PI 값은 절대로 바뀌지 않는 상수 이기 때문이죠. 따라서, 프로그래머가 밤에 졸면서 코딩을 하다가 아래와 같이

```
PI = 10;
```

PI 의 값을 문장을 집어 넣었다고 해도 컴파일 시 오류가 발생하여 프로그래머는 이를 고칠 수 있게 됩니다. 반면에 PI 를 그냥 double 형 변수로 선언했다고 해봅시다.

```
double PI = 3.141592;
```

그렇다면 프로그래머가 아래와 같은 코드를 잠결에 집어 넣었다면

```
PI = 10;
```

컴파일러는 이를 오류로 처리하지 않습니다. 이는 엄청나게 큰일이 아닐 수 없죠. 만일 고객들에게 원의 넓이를 계산하는 프로그램을 만들어 주었는데 잘못해서 이상한 값이 나오면 어떻겠습니까? 물론 위와 같이 간단한 오류는 잡아내기 쉽지만 프로그램이 커지만 커질 수록 위와 같은 오류를 잡아내는 것은 여간 힘든 일이 아닙니다. 따라서, 우리는 '절대로 바뀌지 않을 것 같은 값에는 무조건 const 키워드를 붙여주는 습관' 을 기르는 것이 중요합니다.

아무튼. 이번에는 포인터에도 const 를 붙일 수 있는지 생각해 봅시다.

```
/* 상수 포인터? */
#include <stdio.h>
int main() {
  int a;
  int b;
  const int* pa = &a;

  *pa = 3;  // 올바르지 않은 문장
```

```
  pa = &b;  // 올바른 문장
  return 0;
}
```

컴파일 해보면 오류가 발생합니다.

컴파일 오류

```
error C2166: l-value가 const 개체를 지정합니다.
```

일단, 위 오류가 왜 발생하였는지에 대해 이야기 하기 앞서서 아래 문장이 무슨 의미를 가지는지 살펴 봅시다.

```
const int* pa =
  &a;  // int* pa 와 같이 정의해도 int *pa 와 같다는 사실은 다 알고 있죠?
```

여러분은 위 문장을 보면 다음과 같은 생각이 떠오를 것입니다. "저 포인터는 const int 형을 가리키는 포인터인데, 어떻게 int 형 변수 a 의 주소값이 대입 될 수 있지? 그러면 안되는 거 아니야?". 하지만, 제가 앞에서 강조해 왔듯이 const 라는 키워드는 **이 데이터의 값은 절대로 바뀌면 안된다** 라고 일러주는 키워드라고 하였습니다.

다시 말해, const int a 라는 변수는 그냥 int 형 변수 a 인데 값이 절대로 바뀌면 안되는 변수일 뿐입니다. 따라서, const int a 변수도 그냥 int 형이라 말할 수 있습니다. (다만 '변'수가 아닐 뿐)

따라서 const int* 의 의미는 const int 형 변수를 가리킨다는 것이 아닙니다. int 형 변수를 가리키는데 **그 값을 절대로 바꾸지 말라** 라는 의미이죠. 즉, pa 는 어떠한 int 형 변수를 가리키고 있습니다. 그런데 const 가 붙었으므로 pa 가가리키는 변수의 값은 절대로 바뀌면 안되게 됩니다.

여기서 *pa* 가 라는 부분을 강조한 이유는 a 자체는 변수 이므로 값이 자유롭게 변경될 수 있기 때문입니다. 하지만 pa 를 통해서 a 를 간접적으로 가리킬 때 에는 컴퓨터가 **아, 내가 const 인 변수를 가리키고 있구나** 로 생각하기 때문에(const int* 로 포인터를 정의하였으므로) 값을 바꿀 수 없게 됩니다.

결과적으로 아래의 문장은 오류를 출력합니다.

```
*pa = 3;  // 올바르지 않은 문장
```

물론 a = 3; 과 같은 문장은 오류를 출력하지 않습니다. 앞에서도 말했듯이 변수 a 자체는 const 가 아니기 때문이죠.

```
pa = &b;  // 올바른 문장
```

그렇다면 위 문장은 옳은 문장 입니다. 왜 일까요? (아마 당연하다고 생각하면 여러분은 훌륭한 학생들 입니다) 이는 아래 예제와 함께 설명하도록 하겠습니다.

```c
/* 상수 포인터? */
#include <stdio.h>
int main() {
  int a;
  int b;
  int* const pa = &a;

  *pa = 3;  // 올바른 문장
  pa = &b;  // 올바르지 않은 문장

  return 0;
}
```

역시 컴파일 해보면

컴파일 오류

```
error C2166: l-value가 const 개체를 지정합니다.
```

앞서 보았던 오류와 동일한 오류가 뜹니다. 그런데 위치가 다릅니다. 앞에서는 위 문장에서 오류가 발생했는데 이번엔 아래에서 발생합니다. 일단, 포인터의 정의 부분 부터 이야기 해봅시다.

```c
int* const pa = &a;
```

차근차근 봐 보면, 우리는 `int*` 를 가리키는 `pa` 라는 포인터를 정의하였습니다. 그런데 이번에는 `const` 키워드가 `int*` 앞에 있는 것이 아니라 `int*` 와 `pa` 사이에 놓이고 있습니다. 뭐지? 하지만 이 것은 `const` 키워드의 의미를 그대로 생각해 보면 간단합니다. `pa` 의 값이 바뀐 안된다는 것이지요.

그런데 제일 처음에 포인터를 배울 때 강조했듯이, 포인터에는 가리키는 데이터의 주소값, 즉 위 경우 `a` 의 주소값이 `pa` 저장되는 것이지요. 따라서, 이 `pa` 가 `const` 라는 의미는 `pa` 의 값이 절대로 바뀔 수 없다는 것인데, `pa` 는 포인터가 가리키는 변수의 주소값이 들어 있으므로 결과적으로 `pa` 가 처음에 가리키는 것 (a) 말고 다른 것은 절대로 건드릴 수 없다는 것 입니다.

```
pa = &b; // 올바르지 않은 문장
```

결론적으로 위 문장은 오류를 뿜게 됩니다. 왜냐하면 `pa` 가 다른 변수를 가리키기 때문이죠 (즉 `pa` 에 저장된 주소값을 바꾸므로) 반면에 위의 예제에서 오류가 났던 문장은 올바르게 돌아갑니다.

```
*pa = 3;  // 올바른 문장
```

왜냐하면 pa 가 가리키는 값을 바꾸면 안된다는 말은 안했기 때문이죠. (그냥 int*)

한 번 위에 나와있던 것을 모두 합쳐 보면

```
/* 상수 포인터? */
#include <stdio.h>
int main() {
  int a;
  int b;
  const int* const pa = &a;

  *pa = 3;  // 올바르지 않은 문장
  pa = &b;  // 올바르지 않은 문장

  return 0;
}
```

와 같이 되겠지요. 어때요? 쉽죠?

포인터의 덧셈

이번에는 포인터의 덧셈과 뺄셈에 대해서 다루어 보도록 하겠습니다. 앞에서도 강조하였지만 지금 하는 작업들이 무의미해 보이고 쓸모 없어 보이지만 나중에 정말로 중요하게 다루어 집니다. 조금만 힘내세요 (아마도 C 언어에서 가장 재미 없는 부분일듯.)

```
/* 포인터의 덧셈 */
#include <stdio.h>
int main() {
  int a;
  int* pa;
  pa = &a;

  printf("pa 의 값 : %p \n", pa);
  printf("(pa + 1) 의 값 : %p \n", pa + 1);

  return 0;
}
```

성공적으로 컴파일 해보면

실행 결과

```
pa 의 값 : 0x7ffd6a32fc4c
(pa + 1) 의 값 : 0x7ffd6a32fc50
```

여러분의 출력 결과는 위에 나온 결과와 다를 수 있습니다. 다만, 두 수의 차이는 4 일 것입니다. (16 진수임에 유의하세요; 50 에서 4c 를 빼면 4!)

아마 여러분은 출력된 결과를 보면서 깜짝 놀랐을 것입니다. 우리는 분명히

```
printf("(pa + 1) 의 값 : %p \n", pa + 1);
```

에서 `pa + 1` 의 값을 출력하라고 명시하였습니다. 제가 앞에서도 이야기 하였듯이 `pa` 에는 자신이 가리키는 변수의 주소값이 들어갑니다. 따라서, `pa + 1` 을 하면 0x7ffd6a32fc4c 에 1 이 더해진 0x7ffd6a32fc4d 가 아니라, 4 가 더해진 0x7ffd6a32fc50 이 출력되었습니다. 이게 도대체 무슨 일입니까? `0x7ffd6a32fc4c + 1 = 0x7ffd6a32fc50` 이라고요?

위 해괴한 계산 결과를 해결하기 앞서, 우리는 포인터의 형이 `int*` 라는 것을 알 수 있었습니다. 그런데 `int` 가 4 바이트 이니까...설마?

일단, 위 추측을 확인해보기 위해 `int` 포인터 말고도 크기가 다른 `char` 이다 `double` 등에 대해서도 해봅시다.

```c
/* 과연? */
#include <stdio.h>
int main() {
  int a;
  char b;
  double c;
  int* pa = &a;
  char* pb = &b;
  double* pc = &c;

  printf("pa 의 값 : %p \n", pa);
  printf("(pa + 1) 의 값 : %p \n", pa + 1);
  printf("pb 의 값 : %p \n", pb);
  printf("(pb + 1) 의 값 : %p \n", pb + 1);
  printf("pc 의 값 : %p \n", pc);
  printf("(pc + 1) 의 값 : %p \n", pc + 1);

  return 0;
}
```

성공적으로 컴파일 후 실행해 보면

실행 결과

```
pa 의 값 : 0x7ffcf64a2e04
(pa + 1) 의 값 : 0x7ffcf64a2e08
pb 의 값 : 0x7ffcf64a2e03
(pb + 1) 의 값 : 0x7ffcf64a2e04
pc 의 값 : 0x7ffcf64a2e08
(pc + 1) 의 값 : 0x7ffcf64a2e10
```

여러분의 출력 결과는 위에 나온 결과와 다를 수 있습니다.

우왕. 우리의 예상과 정확하게 맞아 떨어졌습니다. pb 의 경우 1 이 더해졌고, pc 의 경우 8 이 더해졌습니다. 그런데, char 은 1 바이트, double 은 8 바이트 이므로 모두 우리가 예상한 결과와 일치합니다. 놀랍군요. 하지만 머리 한 켠에는 또다른 의문이 남습니다. 왜 하라는 대로 안하고 포인터가 가리키는 형의 크기 만큼 더할까요. 사실 이에 대한 해답은 뒤에 나옵니다.

훌륭한 학생이라면 여러가지 모험을 해볼 것 입니다. 예를 들어 포인터의 뺄셈은 허용되는지, 포인터끼리 더해도 되는지 등등.. 말이죠. 우리도 한 번 궁금증을 해결해 봅시다.

일단 직관적으로 포인터의 뺄셈은 허용될 것 같습니다. 왜냐하면 뺄셈은 본질적으로 덧셈과 다를 바 없기 때문이죠. (1 - 1 = 1 + (-1)) 아무튼 해 보면 덧셈과 유사한 결과가 나타납니다.

```c
/* 포인터 뺄셈 */
#include <stdio.h>
int main() {
  int a;
  int* pa = &a;

  printf("pa 의 값 : %p \n", pa);
  printf("(pa - 1) 의 값 : %p \n", pa - 1);

  return 0;
}
```

성공적으로 컴파일 후 실행 해보면

실행 결과

```
pa 의 값 : 0x7ffe4f4fa47c
(pa - 1) 의 값 : 0x7ffe4f4fa478
```

여러분의 출력 결과는 위에 나온 결과와 다를 수 있습니다. 역시 우리의 예상대로 4 가 빼졌습니다.

```c
/* 포인터끼리의 덧셈 */
#include <stdio.h>
int main() {
  int a;
  int *pa = &a;
  int b;
  int *pb = &b;
  int *pc = pa + pb;

  return 0;
}
```

아마 컴파일 해보면 아래와 같은 오류를 만날 수 있습니다.

컴파일 오류

```
error C2110: '+' : 두 포인터를 더할 수 없습니다.
```

왜 C 에서는 두 포인터끼리의 덧셈을 허용하지 않는 것일까요? 사실, 포인터끼리의 덧셈은 아무런 의미가 없을 뿐더러 필요 하지도 않습니다. 두 변수의 메모리 주소를 더해서 나오는 값은 이전에 포인터들이 가리키던 두 개의 변수와 아무런 관련이 없는 메모리 속의 임의의 지점 입니다. 아무런 의미가 없는 프로그램 상에 상관없는 지점을 말이죠. 무언가, 설명이 불충분한 느낌이 들지만 아무튼 포인터 끼리의 덧셈은 아무런 의미가 없기 때문에 C 언어에선 수행할 수 없습니다. 그렇다면, 포인터에 정수를 더하는 것은 왜 되는 것일까요. 아까도 말했듯이 이에 대해선 아래에서 설명해드리겠습니다.

그런데 한 가지 놀라운 점은 포인터끼리의 뺄셈은 가능하다는 것입니다. 왜 그런지에 대한 설명은 나중에 합시다.

```c
/* 포인터의 대입 */
#include <stdio.h>
int main() {
  int a;
  int* pa = &a;
  int* pb;

  *pa = 3;
  pb = pa;

  printf("pa 가 가리키고 있는 것 : %d \n", *pa);
  printf("pb 가 가리키고 있는 것 : %d \n", *pb);

  return 0;
}
```

성공적으로 컴파일 해보면

실행 결과

```
pa 가 가리키고 있는 것 : 3
pb 가 가리키고 있는 것 : 3
```

와 같이 나옵니다. 뭐 당연한 일이지요.

```
pb = pa;
```

부분에서 pa 에 저장되어 있는 값 (즉, pa 가 가리키고 있는 변수의 주소값) 을 pb 에 대입하였습니다. 따라서 pb 도 pa 가 가리키던 것의 주소값을 가지게 되는 것이지요. 결과적으로 pb 와 pa 모두 a 를 가리키게 됩니다. 주의해야 될 점은 pa 와 pb 가 형이 같아야 한다는 점 입니다. 다시 말해 pa 가 `int*` 면 pb 도 `int*` 여야 합니다. 만일 형이 다르다면 형변환을 해주어야 하는데 이에 대한 이야기는 나중에 합시다.

배열과 포인터

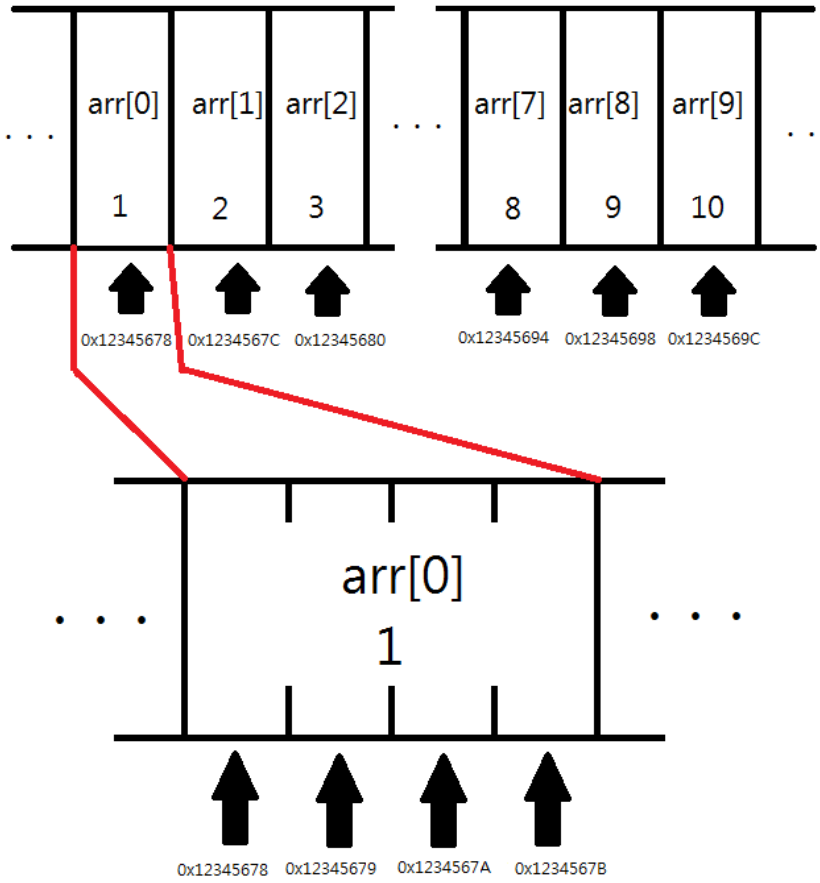
아마 이 단원을 읽다 보면 쇼크를 받을 지도 모르므로 심장이 약하신 분들은 의사와 함께 하십시오.(참고로 저의 경우 많이 놀라서 잠을 잘 못잤습니다)

제가 C 언어를 배우면서 가장 감탄하고도 쇼킹했던 부분이 바로 여기였습니다. 물론, 모든 사람들이 그다지 놀라워 하는 것은 아니지만 저한테는 신선한 충격이였습니다. 아마 이 단원을 배운다면 앞서 '포인터의 연산은 왜 이따구로 하는 거야!' 에 대한 답안을 찾을 수 있을 것 입니다.

이전 강좌에서 (11 강) 저는 여러분에게 배열에 대해 이야기 했었습니다. 기억을 상기해보자면, 배열은 **변수가 여러개 모인 것으로 생각할 수 있다** 라고 이야기 했었지요. 그런데 말이죠. 또다른 놀라운 특징이 있습니다. 바로 배열들의 각 원소는 메모리 상에 연속되게 놓인 다는 점입니다. 뭐, 놀랍지 않다면 말고요. 어쨋든,

```
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

이라는 배열을 정의한다면 메모리 상에서 다음과 같이 나타납니다.



즉, 위와 같이 메모리 상에 연속된 형태로 나타난다는 점이지요. 한 개의 원소는 `int` 형 변수이기 때문에 4 바이트를 차지하게 됩니다. 물론, 위 사실을 믿지 못하시는 분들은 아래와 같이 컴퓨터를 통해 직접 확인해 볼 수 있습니다.

```c
/* 배열의 존재 상태? */
#include <stdio.h>
int main() {
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int i;

  for (i = 0; i < 10; i++) {
    printf("arr[%d] 의 주소값 : %p \n", i, &arr[i]);
  }
  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
arr[0] 의 주소값 : 0x7ffeb5683890
arr[1] 의 주소값 : 0x7ffeb5683894
arr[2] 의 주소값 : 0x7ffeb5683898
arr[3] 의 주소값 : 0x7ffeb568389c
arr[4] 의 주소값 : 0x7ffeb56838a0
arr[5] 의 주소값 : 0x7ffeb56838a4
arr[6] 의 주소값 : 0x7ffeb56838a8
arr[7] 의 주소값 : 0x7ffeb56838ac
arr[8] 의 주소값 : 0x7ffeb56838b0
arr[9] 의 주소값 : 0x7ffeb56838b4
```

와 같이 나타납니다. 여러분의 결과와 주소값은 약간 다를 수 있지만, 어쨋든 4 씩 증가하면 된 것입니다.

아마 여기쯤 왔다면 여러분의 머리를 스쳐지나가는 생각이 들 것입니다! 아! 포인터로도 배열의 원소에 쉽게 접근이 가능하겠구나! (이 생각이 떠오르지 않는 사람은 아마 이 글을 다시 처음부터 읽으셔야 합니다.) 배열의 시작 부분을 가리키는 포인터를 정의한 뒤에 포인터에 1 을 더하면 그 다음 원소를 가리키겠군! 그리고 2 를 더한 그 다음 다음 원소를 가리킨다!!

위와 같은 일이 가능한 이유는 포인터는 자신이 가리키는 데이타의 '형' 의 크기를 곱한 만큼 덧셈을 수행하기 때문이죠. 즉 `p` 라는 포인터가 `int a;` 를 가리킨다면 `p + 1` 을 할 때 `p` 의 주소값에 사실은 `1*4` 가 더해지고, `p + 3` 을 하면 `p` 의 주소값에 `3 * 4` 인 12 가 더해진다는 것입니다.

한 번 이 아이디어를 적용시켜서 배열의 원소를 가리키는 포인터를 만들어봅시다.

```c
/* 과연? */
#include <stdio.h>
int main() {
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int* parr;
  int i;
  parr = &arr[0];

  for (i = 0; i < 10; i++) {
    printf("arr[%d] 의 주소값 : %p ", i, &arr[i]);
    printf("(parr + %d) 의 값 : %p ", i, (parr + i));

    if (&arr[i] == (parr + i)) {
      /* 만일 (parr + i) 가 성공적으로 arr[i] 를 가리킨다면 */
      printf(" --> 일치 \n");
    } else {
      printf("--> 불일치\n");
    }
  }
```

```
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
arr[0] 의 주소값 : 0x7ffedbe31530 (parr + 0) 의 값 : 0x7ffedbe31530
 ↪  --> 일치
arr[1] 의 주소값 : 0x7ffedbe31534 (parr + 1) 의 값 : 0x7ffedbe31534
 ↪  --> 일치
arr[2] 의 주소값 : 0x7ffedbe31538 (parr + 2) 의 값 : 0x7ffedbe31538
 ↪  --> 일치
arr[3] 의 주소값 : 0x7ffedbe3153c (parr + 3) 의 값 : 0x7ffedbe3153c
 ↪  --> 일치
arr[4] 의 주소값 : 0x7ffedbe31540 (parr + 4) 의 값 : 0x7ffedbe31540
 ↪  --> 일치
arr[5] 의 주소값 : 0x7ffedbe31544 (parr + 5) 의 값 : 0x7ffedbe31544
 ↪  --> 일치
arr[6] 의 주소값 : 0x7ffedbe31548 (parr + 6) 의 값 : 0x7ffedbe31548
 ↪  --> 일치
arr[7] 의 주소값 : 0x7ffedbe3154c (parr + 7) 의 값 : 0x7ffedbe3154c
 ↪  --> 일치
arr[8] 의 주소값 : 0x7ffedbe31550 (parr + 8) 의 값 : 0x7ffedbe31550
 ↪  --> 일치
arr[9] 의 주소값 : 0x7ffedbe31554 (parr + 9) 의 값 : 0x7ffedbe31554
 ↪  --> 일치
```

정확히 모두 일치가 나옵니다. 위 소스코드가 이해가 안되는 분들이 있을 까봐 살짝 설명을 드리기는 하겠습니다.

```
parr = &arr[0];
```

`parr` 이라는 `int` 형을 가리키는 포인터는 `arr[0]` 이라는 `int` 형 변수를 가리킵니다. (배열의 각 원소는 하나의 변수로 생각할 수 있다는 사실은 까먹지 않았죠?)

```
printf("arr[%d] 의 주소값 : %p ", i, &arr[i]);
printf("(parr + %d) 의 값 : %p ", i, (parr + i));
```

이제, arr[i] 의 주소값과 (parr + i) 의 값을 출력해봅니다. 만일 parr + i 의 값이 arr[i] 의 주소값과 같다면 하단의 if-else 에서 일치가 출력되고 다르다면 불일치가 출력되게 됩니다. 그런데, 이미 예상하고 있던 바이지만 parr 이 int 형이므로 + i 를 하면 주소값에는 사실상 4*i 가 더해지게 되는 것이지요. 이 때 arr[i] 의 주소값도 i 가 하나씩 커질 때 마다 4 씩 증가하므로 (int 형 배열이므로) 결과적으로 모든 결과가 일치하게 되는 것 입니다.

이렇게 포인터에 정수를 더하는 것 만으로도 배열의 각 원소를 가리킬 수 있습니다. 그렇다면 * 를 이용하여 원소들과 똑같은 역할을 할 수 있게 되겠군요. 마치 아래 예제 처럼 말이지요.

```c
/* 우왕 */
#include <stdio.h>
int main() {
  int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int* parr;

  parr = &arr[0];

  printf("arr[3] = %d , *(parr + 3) = %d \n", arr[3], *(parr + 3));
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
arr[3] = 4 , *(parr + 3) = 4
```

와 같이 동일하게 접근할 수 있게 됩니다.

즉 parr + 3 을 수행하면, arr[3] 의 주소값이 되고, 거기에 * 를 붙여주면 * 의 연산자의 역할이 '그 주소값에 해당하는 데이터를 의미해라!' 라는 뜻이므로 *(parr + 3) 은 arr[3] 과 동일하게 된다는 것입니다. 어때요? 놀랍지요. 포인터의 덧셈이 왜 그렇게 수행되는지 속 시원하게 해결되는 것 같나요?

배열의 이름의 비밀

아마 여러분들 중 대다수는 배열을 처음 배울 때 다음과 같은 실수를 하신 경험이 있을 것 입니다. (나만 그런가?)

```c
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};
```

```
  printf("%d", arr);
}
```

그러곤 1 도, 2 도, 3 도, 아닌 이상한 값이 나오는 것을 보고 당황하셨겠죠. 그런데, 놀랍게도 그 때 출력되는 값은 아래와 같습니다.

```c
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};

  printf("arr 의 정체 : %p \n", arr);
  printf("arr[0] 의 주소값 : %p \n", &arr[0]);

  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
arr 의 정체 : 0x7fff1e868b1c
arr[0] 의 주소값 : 0x7fff1e868b1c
```

와 같이 나옵니다. 와우! 놀랍게도 `arr` 과 `arr[0]` 의 주소값이 동일합니다.

따라서 배열에서 배열의 이름은 배열의 첫 번째 원소의 주소값을 나타내고 있다는 사실을 알 수 있습니다. 그렇다면 배열의 이름이 배열의 첫 번째 원소를 가리키는 포인터라고 할 수 있을까요? **아닙니다!**

주의 사항

이 부분은 (저를 포함한) 많은 사람들이 헷갈렷던 부분들 중 하나 입니다. 포인터를 갓 배운 상태에서 읽어보면 이해가 잘 가지 않을 수 도 있으니, 나중에 포인터와 조금 친숙해진다면 꼭 다시 읽어보는 것을 추천합니다.

배열은 배열이고 포인터는 포인터이다.

예를 들어서 다음과 같이 `sizeof` 를 사용하는 코드를 살펴봅시다. 기억을 상기해보자면 `sizeof` 는 크기를 알려주는 연산자 입니다.

```c
#include <stdio.h>
int main() {
```

```
  int arr[6] = {1, 2, 3, 4, 5, 6};
  int* parr = arr;

  printf("Sizeof(arr) : %d \n", sizeof(arr));
  printf("Sizeof(parr) : %d \n", sizeof(parr));
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Sizeof(arr) : 24
Sizeof(parr) : 8
```

와 같이 나옵니다. 재미 있게도

```
  printf("Sizeof(arr) : %d \n", sizeof(arr));
```

sizeof 를 arr 자체에 그대로 썼을 경우 배열의 **실제 크기** 가 나옵니다. 우리의 arr 배열에는 int 원소 6 개가 있으므로 크기가 24 가 되겠지요. 반면에 parr 에 sizeof 연산자를 사용하였을 경우

```
  printf("Sizeof(parr) : %d \n", sizeof(parr));
```

배열의 자체의 크기가 아니라 그냥 포인터의 크기를 알려줍니다 (64 비트 컴퓨터 이므로 출력된 것 처럼 8 바이트 겠지요).

따라서 배열의 이름과, 첫 번째 원소의 주소값은 엄밀히 **다른 것** 인 것입니다. 그렇다면 도대체 왜 두 값을 출력 했을 때 같은 값이 나왔을까요?

그 이유는 C 언어 상에서 배열의 이름이 sizeof 연산자나 주소값 연산자(&)와 사용될 때 (예를 들어 &arr) 경우를 빼면, **배열의 이름을 사용시 암묵적으로 첫 번째 원소를 가리키는 포인터로 타입 변환되기 때문입니다.**

그렇다면 이제 왜 아래 코드에서 배열의 시작 원소의 주소값이 나왔는지 이해가 가시나요?

```
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};

  printf("arr 의 정체 : %p \n", arr);
  printf("arr[0] 의 주소값 : %p \n", &arr[0]);
```

```
  return 0;
}
```

arr 이 sizeof 랑도, 주소값 연산자랑도 사용되지 않았기에, arr 은 첫 번째 원소를 가리키는 포인터로 타입 변환되었기에, &arr[0] 와 일치하게 됩니다.

[] 연산자의 역할

여러분들 중에서 많은 분들은 [] 가 연산자였다는 사실을 보고 깜짝 놀랐을 것 입니다. 그런데, 4강에서 연산 순위에 대해 이야기 하였을 때 눈썰미가 좋으신 분들은 [] 가 연산자로 나와있음을 보셨을 것입니다.

1	( ) [ ] -> .	왼쪽 우선
2	! ~ ++ -- + -(부호) *(포인터) & sizeof 캐스트	오른쪽 우선
3	*(곱셈) / %	왼쪽 우선
4	+ -(덧셈, 뺄셈)	왼쪽 우선
5	<< >>	왼쪽 우선
6	< <= > >=	왼쪽 우선
7	== !=	왼쪽 우선
8	&	왼쪽 우선
9	^	왼쪽 우선
10	|	왼쪽 우선
11	&&	왼쪽 우선
12	||	왼쪽 우선
13	? :	오른쪽 우선
14	= 복합대입	오른쪽 우선
15	,	왼쪽 우선

www.winapi.com 에서 가져온 자료 입니다.

그런데, 우리는 앞서 포인터 연산이 어떻게 돌아가는지 배웠기 때문에 [] 연산자의 역할을 대충 짐작할 수 있습니다.

```
/* [] 연산자 */
#include <stdio.h>
int main() {
  int arr[5] = {1, 2, 3, 4, 5};

  printf("a[3] : %d \n", arr[3]);
  printf("*(a+3) : %d \n", *(arr + 3));
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
a[3] : 4
*(a+3) : 4
```

음... 이미 앞에서 다룬 내용을 모두 이해했더라면 위 정도쯤은 쉽게 이해할 수 있을 것입니다. 사실 컴퓨터는 C 에서 [] 라는 연산자가 쓰이면 자동적으로 위 처럼 형태로 바꾸어서 처리하게 됩니다. 즉, 우리가 arr[3] 이라 사용한 것은 사실 *(arr + 3) 으로 바뀌어서 처리가 된다는 뜻이지요.

그리고 arr 은 + 연산자와 사용되기 때문에 앞서 말했듯이 **첫 번째 원소를 가리키는 포인터** 로 변환되어서 arr + 3 이 포인터 덧셈을 수행하게 됩니다. 그리고 이는 배열의 4 번째 원소를 가리키게 되겠지요.

따라서 다음과 같이 신기한 연산도 가능합니다.

```c
/* 신기한 [] 사용 */
#include <stdio.h>
int main() {
  int arr[5] = {1, 2, 3, 4, 5};

  printf("3[arr] : %d \n", 3 [arr]);
  printf("*(3+a) : %d \n", *(arr + 3));
  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
3[arr] : 4
*(3+a) : 4
```

3[arr] 은 무언가 조금 이상한 표현 입니다. 사실 이렇게 사용한다면 가독성도 떨어지고 한 번에 이해도 되지 않기에 대부분의 프로그래머들은 arr[3] 으로 사용할 것입니다. 하지만, 앞에서도 [] 는 연산자로 3[arr] 을 *(3+arr) 로 바꿔주기 때문에 arr[3] 과 동일한 결과를 출력할 수 있게 되지요.

포인터의 정의

앞에서 말하기를 int 를 가리키는 포인터를 정의하기 위해 다음의 두 문장을 모두 사용할 수 있다고 했습니다.

```
int* p;
int *p;
```

그런데 말이죠. 제 강좌 말도 다른 곳에서 C 언어를 공부했던 사람들이라면 아래와 같은 형식을 훨씬 많이 쓴다는 사실을 알 수 있었을 것입니다.

```
int *p;
```

왜 일까요? 우리가 int 형 변수를 여러개 한 번에 선언하려 했을 때 int a,b,c,d; 라 하잖아요. 포인터 변수를 여러개 선언 하려면 아래와 같이 해야 합니다.

```
int *p, *q, *r;
```

물론

```
int *p, *q, *r;
```

게 해도 됩니다. 다만,

```
int* p;
```

꼴로 한다면 다음과 같이 실수 할 확률이 매우 커지게 됩니다. 왜냐하면 아래와 같이 한다면

```
int *p, q, r;
```

p 만 int 를 가리키는 포인터 이고, q, r 은 평범한 int 형 변수가 됩니다. 따라서, 앞으로 저는 제 강좌에서 모든 포인터들은

```
int *p;
```

꼴로 선언 하도록 하겠습니다.

생각해 볼 문제

문제 1

`int arr[3][3];` 과 같은 배열은 내부적으로 어떻게 처리되는지 생각해보세요 (난이도 : 中)

문제 2

`int* arr[3];` 과 같은 배열이 가지는 의미는 무엇일까요? (난이도 : 中)

포인터의 포인터

안녕하세요 여러분~! 이전 강좌는 잘 보시고 계시는지요? 아마도 이번 강좌가 최대의 난관일 듯 하네요. 이번 강좌를 잘 이해하냐, 이해 못하냐에 따라서 C 언어가 쉽다/어렵다가 완전히 좌우됩니다. 그러니 지금 졸린 사람들은 잠을 자고 쌩쌩할 때 오시길 바랍니다. (아마도 이 부분이 C 언어에서 가장 어려울 부분이 될 듯 하네요.. 저도 최대한 쉽게 설명하기 위해 노력하겠습니다)

잠깐 지난 시간에 배웠던 것을 머리속으로 상기시켜봅시다. 일단,

배열을 배열이고 포인터는 포인터이다. 다만;

- `sizeof` 와 주소값 연산자와 함께 사용할 때를 제외하면, 배열의 이름은 첫 번째 원소를 가리킨다.
- `arr[i]` 와 같은 문장은 사실 컴파일러에 의해 `*(arr + i)` 로 변환된다.

이 두 가지 사실을 머리속에 잘 들어 있겠지요. 만일 위 두 문장을 읽으면서 조금이라도 의구심이 드는 사람은 바로 뒤로가기를 눌러서 이전 강좌를 보시기 바랍니다.

1 차원 배열 가리키기

일단, 강의의 시작은 간단한 것으로 해보겠습니다. 이전해도 말했듯이 `int arr[10];` 이라는 배열을 만든다면 앞서 이야기한 두 가지 경우를 제외한다면 `arr` 이 `arr[0]` 을 가리키는 포인터로 타입 변환 된다고 하였습니다.

그렇다면 다른 `int*` 포인터가 이 배열을 가리킬 수 있지 않을까요? 한 번 프로그램을 짜봅시다.

```c
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};
  int *parr;

  parr = arr;
  /* parr = &arr[0]; 도 동일하다! */

  printf("arr[1] : %d \n", arr[1]);
  printf("parr[1] : %d \n", parr[1]);
  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
arr[1] : 2
parr[1] : 2
```

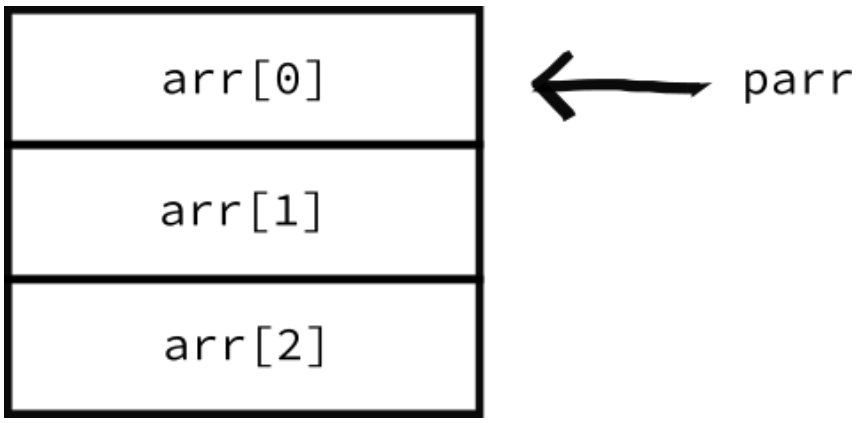
일단, 중점적으로 볼 부분은 아래와 같습니다.

```
parr = arr;
```

바로 arr 를 parr 에 대입하는 부분이지요. 앞에서 말했듯이 arr 은 배열의 첫 번째 원소를 가리키는 포인터로 변환되고, 그 원소의 타입이 int 이므로, 포인터의 타입은 int* 가 되겠지요. 위 문장은 아래와 정확히 동일한 문장이 됩니다.

```
parr = &arr[0]
```

따라서, parr 을 통해서 arr 을 이용했을 때와 동일하게 배열의 원소에 마음껏 접근할 수 있게 되는 것이 됩니다. 위 모습을 한 번 그림으로 나타내보면 (아마도 여러분들은 지금 수준이라면 머리속으로 다 그릴 수 있어야 할 것입니다)



참고적으로 한 방의 크기는 그림의 단순화를 위해 4 바이트로 하였습니다.

```c
/* 포인터 이용하기 */
#include <stdio.h>
int main() {
  int arr[10] = {100, 98, 97, 95, 89, 76, 92, 96, 100, 99};

  int* parr = arr;
  int sum = 0;

  while (parr - arr <= 9) {
    sum += (*parr);
    parr++;
```

```
  }

  printf("내 시험 점수 평균 : %d \n", sum / 10);
  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
내 시험 점수 평균 : 94
```

일단, 포인터를 이용한 간단한 예제를 다루어보겠습니다.

```
int* parr = arr;
```

먼저, int 형 1 차원 배열을 가리킬 수 있는 int* 포인터를 정의하였습니다. 그리고, 이 parr 은 배열 arr 을 가리키게 됩니다.

```
while (parr - arr <= 9) {
  sum += (*parr);
  parr++;
}
```

그 다음 while 문을 살펴봅시다. while 문을 오래전에 배워서 기억이 안난다면 여기로 돌아가세요!

이 while 문은 parr - arr 이 9 이하 일 동안 돌아가게 됩니다. sum 에 parr 이 가리키는 원소의 값을 더했습니다. += 연산자의 의미는 아시죠? sum += (*parr); 문장은 sum = sum + *parr 와 같다는 것 알고 계시지요?

```
parr++;
```

parr 을 1 증가시켰습니다. 이전 강좌에서도 이야기 하였지만 포인터 연산에서 1 증가시킨다면, parr 에 저장된 주소값에 1 이 더해지는 것이 아니라 1 * (포인터가 가리키는 타입의 크기) 가 더해진다는 것이지요.

즉, int 형 포인터 이므로 4 가 더해지게되서, 배열의 그 다음 원소를 가리킬 수 있게 됩니다. 아무튼, 위 작업을 반복하면 arr 배열의 모든 원소들의 합을 구하게 됩니다. while 문에서 9 이하일 동안만 반복하는 이유는, parr - arr >= 10 이 된다면 parr[10 이상의 값] 을 접근하게 되므로 오류를 뿜게 됩니다.

여기서 궁금한 것이 없나요? 우리가 왜 굳이 `parr` 을 따로 선언하였을까요? 우리는 `arr` 이 `arr[0]` 을 가리킨다는 사실을 알고 있으므로 `arr` 을 증가시켜서 `*(arr)` 으로 접근해도 되지 않을까요? 한 번, `arr` 의 값을 변경할 수 있는지 없는지 살펴봅시다.

```c
/* 배열명 */
#include <stdio.h>
int main() {
  int arr[10] = {100, 98, 97, 95, 89, 76, 92, 96, 100, 99};

  arr++;  // 오류
  return 0;
}
```

컴파일 해보면

컴파일 오류

```
error C2105: '++'에 l-value가 필요합니다.
```

와 같은 오류를 만나게 됩니다.

배열의 이름이 첫 번째 원소를 가리키는 포인터로 타입 변경 된다고 했을 때, 이는 단순히 배열의 첫 번째 원소를 가리키는 주소값 자체가 될 뿐입니다. 따라서 `arr++` 문장은 C 컴파일러 입장에서 다음을 수행한 것과 같습니다.

```c
(0x7fff1234) ++;
```

이는 애초에 말이 안되는 문장 이지요.

포인터의 포인터

똑똑한 분들이라면 이러한 것들에 대해서도 생각해 보신 적이 있을 것입니다. 물론, 안하셔도 상관 없고요. 저의 경우 포인터 처음 배울 때 그것 마저 이해하기도 힘들어서 한참 버벅거렸습니다 :) 아무튼. 지금 머리속으로 예상하시는 대로 포인터의 포인터는 다음과 같이 정의합니다.

```c
int **p;
```

위는 **`int` 를 가리키는 포인터를 가리키는 포인터** 라고 할 수 있습니다. 쉽게 머리에 와닿지 않죠? 당연합니다. 이전 강좌의 내용도 어려워 죽겠는데 위 내용까지 머리속에 쑤셔 넣으려면 얼마나 힘들겠어요? 그래서, 한 번 예제를 봅시다.

```c
/* 포인터의 포인터 */
#include <stdio.h>
int main() {
  int a;
  int *pa;
  int **ppa;

  pa = &a;
  ppa = &pa;

  a = 3;

  printf("a : %d // *pa : %d // **ppa : %d \n", a, *pa, **ppa);
  printf("&a : %p // pa : %p // *ppa : %p \n", &a, pa, *ppa);
  printf("&pa : %p // ppa : %p \n", &pa, ppa);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
a : 3 // *pa : 3 // **ppa : 3
&a : 0x7ffd26a79dd4 // pa : 0x7ffd26a79dd4 // *ppa :
↪  0x7ffd26a79dd4
&pa : 0x7ffd26a79dd8 // ppa : 0x7ffd26a79dd8
```

여러분의 결과는 약간 다를 수 있습니다. 다만, 같은 행에 있는 값들이 모두 같음을 주목하세요

일단 위에 보시다 싶이 같은 행에 있는 값들은 모두 같습니다. 사실, 위 예제는 그리 어려운 것이 아닙니다. 포인터에 제대로 이해만 했다면 말이죠. 일단 ppa 는 `int*` 를 가리키는 포인터 이기 때문에

```c
ppa = &pa;
```

와 같이 이전의 포인터에서 했던 것 처럼 똑같이 해주면 됩니다. ppa 에는 pa 의 주소값이 들어가게 되죠.

```c
printf("&pa : %p // ppa : %p \n", &pa, ppa);
```

따라서 우리는 위의 문장이 같은 값을 출력함을 알 수 있습니다. 위의 실행한 결과를 보아도 둘다 1636564 를 출력했잖아요.

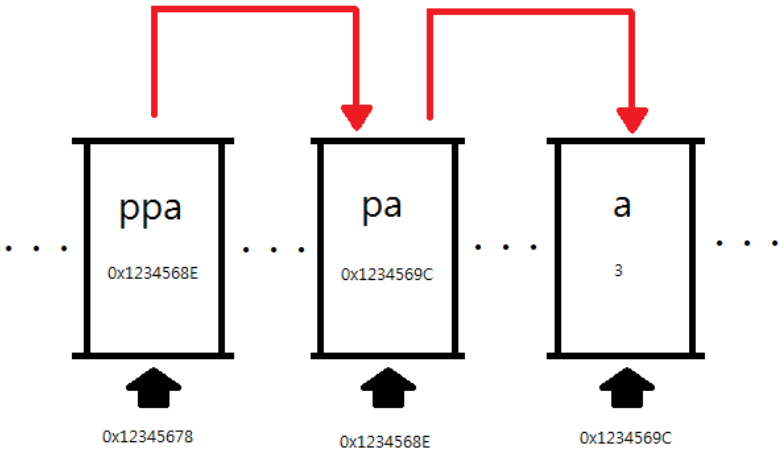
```
printf("&a : %p // pa : %p // *ppa : %p \n", &a, pa, *ppa);
```

그리고 이제 아래에서 두 번째 문장을 봐 봅시다. pa 가 a 를 가리키고 있으므로 pa 에는 a 의 주소값이 들어갑니다. 따라서, &a 와 pa 는 같은 값이 되겠지요. 그러면 *ppa 는 어떨까요? ppa 가 pa 를 가리키고 있으므로 *ppa 를 하면 pa 를 지칭하는 것이 됩니다. 따라서 역시 pa 의 값, 즉 &a 의 값이 출력되게 됩니다.

```
printf("a : %d // *pa : %d // **ppa : %d \n", a, *pa, **ppa);
```

마지막으로 위의 문장을 살펴 봅시다. pa 가 a 를 가리키고 있으므로 *pa 를 하면 a 를 지칭하는 것이 되어 a 의 값이 출력됩니다. 그렇다면 **ppa 는 어떨까요? 이를 다시 써 보면 *(*ppa) 가 되는데, *ppa 는 pa 를 지칭하는 것이기 때문에 *pa 가 되서, 결국 a 를 지칭하는 것이 됩니다. 따라서, 역시 a 의 값이 출력되겠지요. 어때요? 간단하죠?

위 관계를 그림으로 그리면 다음과 같습니다.



한 방의 크기는 4 byte 이지만 그림 공간의 절약을 위해 한 칸으로 나타내었다.
포인터도 역시 변수 이므로 메모리 상의 일부분을 차지하며, 고유의 메모리 주소가 있다.

배열 이름의 주소값?

지난 강좌에서 배열 이름에 sizeof 연산자와 주소값 연산자를 사용할 때 빼고는 전부다 포인터로 암묵적 변환이 이루어진다고 하였습니다. 그렇다면 주소값 연산자를 사용하면 어떻게 되길래 그러는 것일까요? 한 번 코드로 살펴봅시다.

```
#include <stdio.h>

int main() {
```

```
  int arr[3] = {1, 2, 3};
  int (*parr)[3] = &arr;

  printf("arr[1] : %d \n", arr[1]);
  printf("parr[1] : %d \n", (*parr)[1]);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
arr[1] : 2
parr[1] : 2
```

와 같이 잘 나옵니다.

```
int (*parr)[3] = &arr;
```

&arr 은 도대체 무슨 의미를 가질까요? 이전에 arr 은 int * 로 암묵적 변환된다고 하였으니까 &arr 은 int ** 가 되는 것일까요? **아닙니다!!** 암묵적 변환은 주소값 연산자가 왔을 때에는 이루어지지 않습니다.

arr 이 **크기가 3 인 배열** 이기 때문에, &arr 을 보관할 포인터는 **크기가 3 인 배열을 가리키는 포인터**가 되어야 할 것입니다. 그리고 C 언어 문법상 이를 정의하는 방식은 위와 같습니다.

참고로 parr 을 정의할 때 *parr 을 꼭 () 로 감싸야만 하는데, 만일 괄호를 빼버린다면

```
int *parr[3]
```

와 같이 되어서 C 컴파일러가 int * 원소 3 개를 가지는 배열을 정의한 것으로 오해하게 됩니다 (아래 **포인터의 배열** 에서 좀 더 자세히 다룹니다).

```
printf("parr[1] : %d \n", (*parr)[1]);
```

parr 은 크기가 3 인 배열을 가리키는 포인터 이기 때문에 배열을 직접 나타내기 위해서는 * 연산자를 통해 원래의 arr 을 참조해야 합니다. 따라서 (*parr)[1] 과 arr[1] 은 같은 문장이 되겠지요.

한 가지 재미있는 점은 parr 과 arr 은 같은 값을 가진다는 점입니다.

```c
#include <stdio.h>

int main() {
  int arr[3] = {1, 2, 3};
  int(*parr)[3] = &arr;

  printf("arr : %p \n", arr);
  printf("parr : %p \n", parr);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
arr : 0x7ffda08cd25c
parr : 0x7ffda08cd25c
```

와 같이 나옵니다. arr 과 parr 모두 배열의 첫 번째 원소의 주소값을 출력합니다. 물론 두 개의 타입은 다르지만요. 이는 당연한데, arr 자체가 어떤 메모리 공간에 존재하는 것이 아니기 때문입니다.

이와 같이 C 언어가 변태적으로 동작하는 이유는 사실 그 역사에 숨어있습니다. C 언어는 B 언어에서 파생된 언어인데, B 언어에서는 실제 배열이 있고, 배열을 가리키는 포인터가 따로 있었습니다. B 언어에서 arr 과 arr[0], arr[1] 은 각기 다른 메모리를 차지하는 녀석들이고, arr 이 실제로 arr[0] 를 가리키는 포인터 였습니다. 따라서 arr 의 값을 출력하면 실제로 arr[0] 의 주소값이 나왔고, &arr 은 arr 의 주소값이 나왔겠지요. 따라서 B 언어에서 arr 과 &arr 은 다른 값을 출력했을 것입니다.

하지만 C 언어를 만든 데니스 리치 아저씨는 B 언의 문법을 계승하되, 이와 같이 비효율적으로 배열을 정의할 때 배열의 시작점을 가리키는 포인터로 공간을 낭비하고 싶지 았습니다. 따라서 위와 같이 조금 이상하지만 그래도 메모리 공간을 효율적으로 쓰게 되는 배열 - 포인터 관계가 탄생하게된 것입니다.[3)]

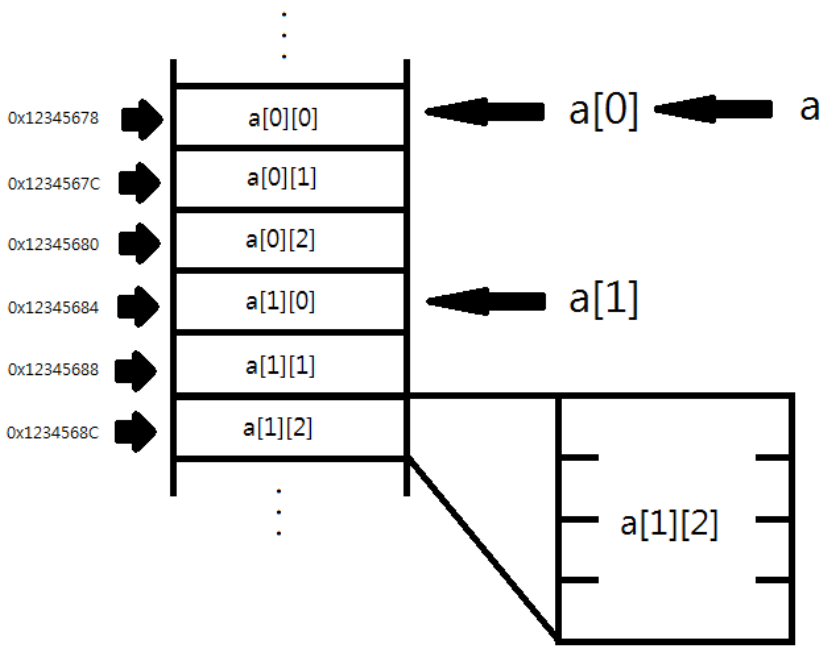
3) C 언어가 C 언어인 이유는 Computer 의 C 를 딴 것이 아니라 그냥 B 언어 다음에 나와서 그렇습니다. 그렇다면 B 언어는 A 언어 다음 언어라서 그런거냐 라고 물을 수 있는데, 사실은 BCPL 이라는 언어를 바탕으로 만들어졌기에 B 언어로 붙여졌습니다. 그럼 마지막으로 BCPL 은 어디서 온거냐 라고 물을 수 있는데 BCPL 은 Basic Computer Programming Language 의 약자로 1966년에 등장한 컴퓨터 언어 입니다. BCPL 의 C 는 진짜 Computer 의 C 네요!

2 차원 배열의 [] 연산자

2 차원 배열의 [] 연산자에 대해선 제가 지난번 강좌에서 생각 해보기 문제로 내었던 것 같은데, 생각해보셨는지요? 일단 이전의 기억을 더듬에서 다음과 같은 배열이 컴퓨터 메모리 상에 어떻게 표현되는지 생각 해보도록 합시다.

```
int a[2][3];
```

물론, 이 배열은 2 차원 배열이므로 평면위에 표시된다고 생각할 수 도 있지만, 컴퓨터 메모리 구조는 1 차원 적이기 때문에 1 차원으로 바꿔서 생각해봅시다. 되었나요? 그렇다면, 제가 그림을 보여드리죠.



실제로 프로그램을 짜서 실행해 보면 메모리 상에 위와 같이 나타남을 알 수 있습니다. 한 번 해보세요~

일단, 위 그림에서 왼쪽에 메모리 상의 배열의 모습이 표현된 것은 여러분이 쉽게 이해하실 수 있으리라 믿습니다. 다만, 제가 설명해야 할 부분은 오른쪽에 큼지막하게 화살표로 가리키고 있는 부분이지요. 먼저 아래의 예제를 봅시다.

```
/* 정말로? */
#include <stdio.h>
int main() {
  int arr[2][3];

  printf("arr[0] : %p \n", arr[0]);
```

```
  printf("&arr[0][0] : %p \n", &arr[0][0]);

  printf("arr[1] : %p \n", arr[1]);
  printf("&arr[1][0] : %p \n", &arr[1][0]);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
arr[0] : 0x7ffda354e530
&arr[0][0] : 0x7ffda354e530
arr[1] : 0x7ffda354e53c
&arr[1][0] : 0x7ffda354e53c
```

표현된 주소값은 여러분과 다를 수 있습니다.

`arr[0]` 의 값이 `arr[0][0]` 의 주소값과 같고, `arr[1]` 의 값이 `arr[1][0]` 의 주소값과 같습니다. 이것을 통해 알 수 있는 사실은 기존의 1 차원 배열과 마찬가지로 `sizeof` 나 주소값 연산자와 사용되지 않을 경우, **`arr[0]` 은 `arr[0][0]` 을 가리키는 포인터로 암묵적으로 타입 변환되고, `arr[1]` 은 `arr[1][0]` 을 가리키는 포인터로 타입 변환된다**라는 뜻이 되겠지요.

주의 사항

1 차원 배열 `int arr[]` 에서 `arr` 과 `&arr[0]` 는 그 자체로는 완전히 다른 것이였던 것처럼 2 차원 배열 `int arr[][]` 에서 `arr[0]` 과 `&arr[0][0]` 와 다릅니다. 다만 암묵적으로 타입 변환 시에 같은 것으로 변할 뿐입니다.

따라서 `sizeof` 를 사용하였을 경우 2 차원 배열의 열의 개수를 계산할 수 있습니다.

```
int main() {
  int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
  printf("전체 크기 : %d \n", sizeof(arr));
  printf("총 열의 개수 : %d \n", sizeof(arr[0]) / sizeof(arr[0][0]));
  printf("총 행의 개수 : %d \n", sizeof(arr) / sizeof(arr[0]));
}
```

성공적으로 컴파일 하였다면

실행 결과

```
전체 크기 : 24
총 열의 개수 : 3
총 행의 개수 : 2
```

와 같이 나옵니다. 먼저 전체 배열에 `sizeof` 를 할 경우 당연하게도 배열의 전체 크기가 나오게 됩니다. 그렇다면

```
printf("총 열의 개수 : %d \n", sizeof(arr[0]) / sizeof(arr[0][0]));
```

위 문장에서 `sizeof(arr[0])` 를 하면 무엇이 나올까요? 바로 0 번째 행의 길이 (총 열의 개수) 가 나오겠지요. 앞에서도 강조해왔듯이 `sizeof` 연산자의 경우 포인터로 타입 변환을 시키지 않기 때문에 `sizeof(arr[0])` 는 마치 `sizeof` 에 1 차원 배열을 전달한 것과 같습니다. 그리고 그 크기 (3) 을 알려주겠지요.

그리고 `sizeof(arr[0][0])` 을 하게 된다면 `int` 의 크기인 4 를 리턴하게 되어서 총 열의 개수를 알 수 있게 됩니다.

```
printf("총 행의 개수 : %d \n", sizeof(arr) / sizeof(arr[0]));
```

그리고 총 행의 개수는 당연히도 전체 크기를 열의 크기로 나눈 것이 됩니다.

이 때, `arr[0][0]` 의 형이 `int` 이므로 `arr[0]` 은 `int*` 형이 되겠고, 마찬가지로 `arr[1]` 도 `int*` 형이 되겠습니다.

자 그렇다면 한 가지 질문을 해보겠습니다. 만일 2 차원 배열의 이름을 포인터에 전달하기 위해서는 해당 포인터의 타입이 뭐가 될까요? `arr[0]` 는 `int *` 가 보관할 수 있으니까, `arr` 은 `int **` 이 보관할 수 있을까요?

> 당연하지. 너가 위에서 설명했잖아. `int*` 를 가리키는 포인터는 `int**` 이라고

그런데 답은 **아니오** 입니다.

포인터의 형(type) 을 결정짓는 두 가지 요소

먼저 포인터의 형을 결정하는 두 가지 요소에 대해 이야기 하기 전에, 위에서 배열의 이름이 왜 `int**` 형이 될 수 없는지에 대해 먼저 이야기 해봅시다. 만일 `int**` 형이 될 수 있다면 맨 위에서 했던 것처럼 `int**` 포인터가 배열의 이름을 가리킨다면 배열의 원소에 자유롭게 접근할 수 있어야만 할 것입니다.

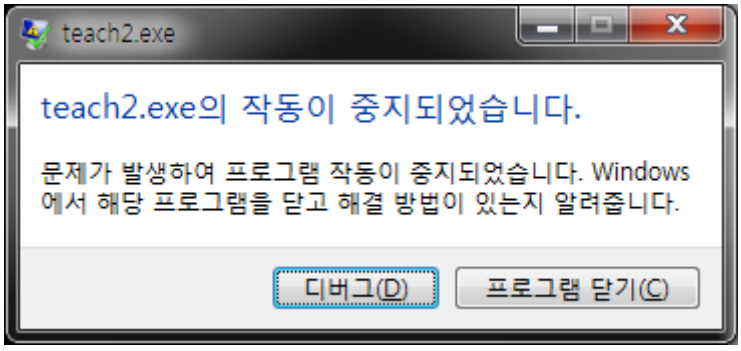
```c
/* 과연 될까? */
#include <stdio.h>
int main() {
  int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
  int **parr;

  parr = arr;

  printf("arr[1][1] : %d \n", arr[1][1]);
  printf("parr[1][1] : %d \n", parr[1][1]);

  return 0;
}
```

그런데 컴파일 시에 아래와 같은 경고가 기분을 나쁘게 하네요.

컴파일 오류

```
warning C4047: '=' : 'int **'의 간접 참조 수준이 'int (*)[3]'과(와)
 ↪  다릅니다.
```

아무튼, 무시하고 실행해봅시다.



헉! 예전에 보았던 친근한 오류가 뜹니다. 무슨 뜻일까요? 예전에 배열에 대해 공부하였을 때 (11 - 1 강) **초기화 되지 않은 값** 에 대해서 이야기한 적이 있었을 것입니다. 이 때, `int arr[3];` 이라 했는데 `arr[10] = 2;` 와 같이 허가되지 않은 공간에 접근하기만 해도 위와 같은 오류가 발생한다고 했습니다.

위 예제의 경우도 마찬가지 입니다. `parr[1][1]` 에서 이상한 메모리 공간의 값에 접근하였기에 발생한 일이지요. 그렇다면 왜? 왜? 이상한 공간에 접근하였을까요?

먼저, 일차원 배열에서 배열의 형과, 시작 주소값을 안다고 칠 때, `n` 번째 원소의 시작 주소값을 알아내는 공식을 생각해봅시다. 만일 이 배열의 형을 `int` 로 가정하고, 시작 주소를 `x` 라고 할 때,

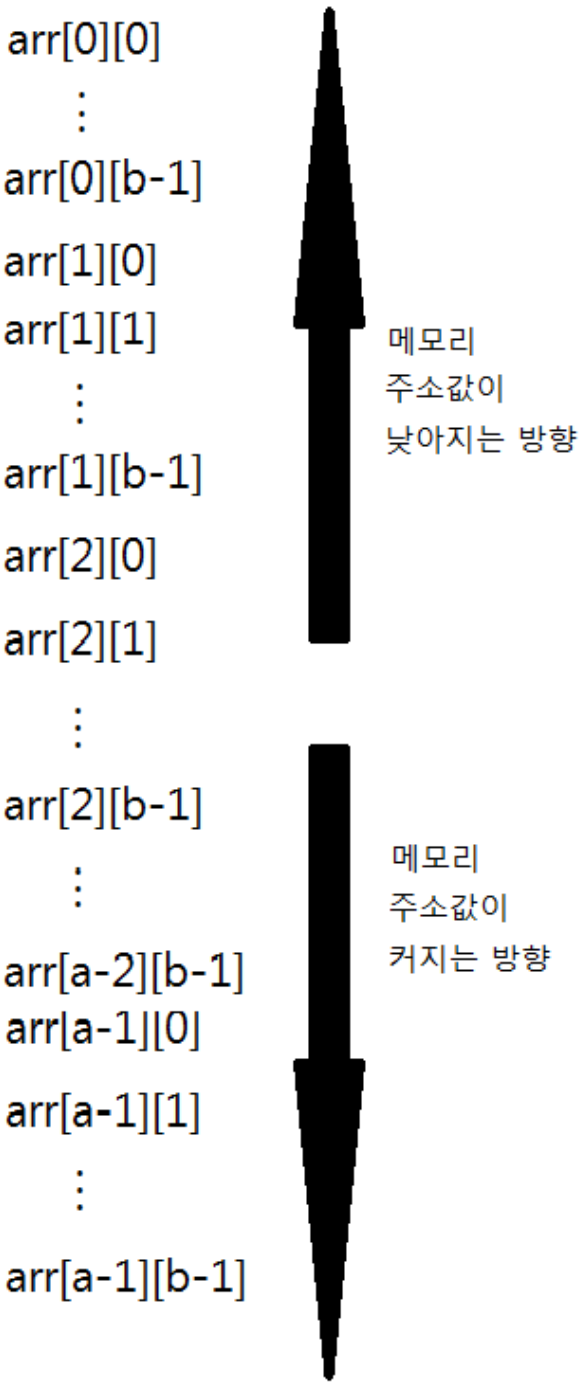
(참고적으로 다 아시겠지만 `int` 는 4 바이트 `n` 번째 원소에 접근한다면 (`x + 4 * (n - 1)` 로 나타낼 수 있죠)

와 같이 나타낼 수 있습니다. 왜냐구요? 아마, 여러분들이 스스로 생각해보세요 :)

이번에는 이차원 배열을 나타내봅시다. 이 이차원 배열이 `int arr[a][b];` 라고 선언되었다면 (여기서 `a` 와 `b` 는 당연히 정수겠죠) 아래와 같이 2차원 평면에 놓여 있다고 생각할 수 있습니다.

arr[0][0]	arr[0][1]	···	arr[0][b-2]	arr[0][b-1]
arr[1][0]	arr[1][1]	···	arr[1][b-2]	arr[1][b-1]
arr[2][0]	arr[2][1]	···	arr[2][b-2]	arr[2][b-1]
⋮	⋮		⋮	⋮
arr[a-1][0]	arr[a-1][1]	···	arr[a-1][b-1]	arr[a-1][b-1]

참고적으로 행은 '가로' 이고, 열은 '세로' 입니다.메모리는 선형(1차원) 이므로 절대로 위와 같이 배열될 일은 없겠지요. 위 이차원 배열을 메모리에 나타내기 위해서는 각 행 부터 읽어주면 됩니다. 즉, 위 배열은 아래와 같이 메모리에 배열됩니다.



사실 위에서도 비슷한 그림이 나오지만 또 그린 이유는 머리에 완전히 박아 두라는 의미 입니다. 즉, `arr[0][0]` 부터 `arr[0][1] .... arr[a-1][b-1]` 순으로 저장되게 되지요. 그렇다면 위 배열의 시작주소를 `x` 라 하고, `int` 형 배열이고, `arr[c][d]` 라는 원소에 접근한다고 칩시다. 그렇다면 이 원소의 주소값은 어떻게 계산될까요?

일단, 위 원소는 (c+1) 번째 행의 (d+1) 번째 열에 위치해 있다고 생각할 수 있습니다. (예를 들어서 arr[0][2] 는 1 번째 행의 3 번째 열에 위치해 있다) 그러면, 먼저 (c+1) 번째 행의 시작 주소를 계산해봅시다. 간단히 생각해보아도 x + c * b * 4 라는 사실을 알 수 있습니다. 왜냐하면 4 를 곱해준 것은 int 이기 때문이고, (c+1) 행 앞에 c 개의 행들이 있는데, 각 행들의 길이가 b 이기 때문이죠.

그리고 이 원소가 (d+1) 번째에 있다는 사실을 적용하면 ((c+1) 행 시작주소) + d * 4 라고 계산될 수 있습니다. 결과적으로 (x + 4 * b * *c + 4 * d가 됩니다)

가 됩니다. 참고적으로 이야기 하자면, 수학에서 곱하기 기호가 매우 자주 등장하므로 생각하는 경향이 있는데, 저도 매번 곱하기 기호를 쓰기 불편하므로 생략하도록 하겠습니다. 위 식은 아래의 식과 동일합니다. (x + 4bc + 4d 로 간추립니다)

주목할 점은 식에 b 가 들어간다는 것입니다. (1 차원 배열에서는 배열의 크기에 관한 정보가 없어도 배열의 원소에 접근할 수 있었는데 말이 1 차원 배열에서는 배열의 크기에 관한 정보가 없어도 배열의 원소에 접근할 수 있었는데 말이죠)

다시 말해, 처음 배열 arr[a][b] 를 정의했을 때의 b 가 원소의 주소값을 계산하기 위해 필요하다는 것입니다. 우리는 이전의 예제에서 int** 로 배열의 이름을 나타낼 수 있다고 생각하였습니다. 하지만 이렇게 선언된 parr 으로 컴퓨터가 parr[1][1] 원소를 참조하려고 하면 컴퓨터는 b 값을 알 수 없기 때문에 제대로된 연산을 수행할 수 없게됩니다.

따라서, 이차원 배열을 가리키는 포인터는 반드시 b 값에 대한 정보를 포함하고 있어야 합니다.

결론적으로 포인터 형을 결정하는 것은 다음 두 가지로 요약할 수 있습니다.

1. 가리키는 것에 대한 정보 (예를 들어, int* 이면 int 를 가리킨다, char** 이면 char* 을 가리킨다 등등)
2. 1 증가시 커지는 크기 (2 차원 배열에서는 b * (형의 크기) 를 의미한다 1 증가시 커지는 크기 (2 차원 배열에서는 b * (형의 크기) 를 의미한다)

여기서 1 증가시 커지는 크기가 2 차원 배열에서는 b * (형의 크기) 를 의미하는지 궁금한 사람들이 있을 것입니다. 한 번 해봅시다.

```c
/* 1 증가하면 ? */
#include <stdio.h>
int main() {
  int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};

  printf("arr : %p , arr + 1 : %p \n", arr, arr + 1);

  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
arr : 0x7fff6fc142b0 , arr + 1 : 0x7fff6fc142bc
```

16 진수의 연산과 친숙하지 않더라도 `0x7fff6fc142bc - 0x7fff6fc142b0` (참고로 예전에도 이야기 했듯이 제 강좌에서 16 진수로 나타내었다는 사실을 명시하기 위해 앞에 `0x` 를 붙인다고 했습니다.) 를 계산해 보면 `0xC` 가 나옵니다. `0xC` 는 십진수로 12 입니다. 근데, 위 배열의 `b` 값은 3 이고 `int` 의 크기는 4 바이트 이므로, `3 * 4 = 12` 가 딱 맞게 되는 것이지요.

왜 그럴까요? 사실, 그 이유는 단순합니다. 거의 맨 위의 그림을 보면 이차원 배열에서 `a` 가 `a[0]` 을 가리키고 있는 그림을 볼 수 있습니다. 만일 1 차원 배열 `b[3]` 이 있을 때 `b + 1` 을 하면 `b[1]` 을 가리키잖아요? 2 차원 배열도 동일하게 `a` 가 1 증가하면 `a[1]` 을 가리키게 됩니다. 다시 말해 두 번째 행의 시작 주소값을 가리키는 포인터를 가리키게 된다는 것이지요.

```c
/* 드디어! 배우는 배열의 포인터 */
#include <stdio.h>
int main() {
  int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
  int(*parr)[3];  // 괄호를 꼭 붙이세요

  parr = arr;  // parr 이 arr 을 가리키게 한다.

  printf("parr[1][2] : %d , arr[1][2] : %d \n", parr[1][2], arr[1][2]);

  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
parr[1][2] : 6 , arr[1][2] : 6
```

드디어, 2 차원 배열을 가리키는 포인터에 대해 이야기 하겠습니다. 2 차원 배열을 가리키는 포인터는 배열의 크기에 관한 정보가 있어야 한다고 했습니다. 2 차원 배열을 가리키는 포인터는 아래와 같이 써주면 됩니다.

```c
/* (배열의 형) */ (*/* (포인터 이름) */)[/* 2 차원 배열의 열 개수 */];
// 예를 들어서
int (*parr)[3];
```

이렇게 포인터를 정의하였을 때 앞서 이야기한 포인터의 조건을 잘 만족하는지 보도록 합시다. 일단, (배열의 형) 을 통해서 원소의 크기에 대한 정보를 알 수 있습니다. 즉, 가리키는 것에 대한 정보를 알 수 있게 됩니다. (조건 1 만족).

또한, [2 차원 배열의 열 개수] 를 통해서 1 증가시 커지는 크기도 알게 됩니다. 바로 배열의 형 크기 - 예를 들어 int 는 4, char 은 1 * (2 차원 배열의 열 개수) 만큼 커지게 됩니다.

```
int (*parr)[3];
```

위와 같이 정의한 포인터 parr 을 해석해 보면, int 형 이차원 배열을 가리키는데, 그 배열의 열의 개수가 3 개 이군요! 라는 사실을 알 수 있습니다 (정확히 말하면, int* 를 가리키는데, 1 증가시 3 이 커진다 라는 의미입니다)

그런데 말이죠. 어디서 위와 같은 형태의 포인터 정의를 보지 않으셨나요? 맞습니다. 저 parr 은 사실 **크기가 3 인 배열을 가리키는 포인터** 를 의미합니다. 그런데 이게 말이 되는게, 1 차원 배열에서 배열의 이름이 첫 번째 원소를 가리키는 포인터로 타입 변환이 된 것 처럼, 2 차원 배열에서 배열의 이름이 첫 번째 **행** 을 가리키는 포인터로 타입 변환이 되어야 합니다. 그리고 그 첫 번째 행은 사실 크기가 3 인 1 차원 배열이지요! 뭔가 아다리가 맞는게 보이시나요?

```
/* 배열 포인터 */
#include <stdio.h>
int main() {
  int arr[2][3];
  int brr[10][3];
  int crr[2][5];

  int(*parr)[3];

  parr = arr;  // O.K
  parr = brr;  // O.K
  parr = crr;  // 오류!!!!

  return 0;
}
```

위 코드에서 parr 이 arr 과 brr 은 받을 수 있어도 crr 은 왜 못받는지 아실 수 있겠죠?

포인터 배열

포인터 배열, 말그대로 **포인터들의 배열** 입니다. 위에서 설명한 배열 포인터는 배열을 가리키는 포인터 였죠. 두 용어가 상당히 헷갈리는데, 그냥 **언제나 진짜는 뒷부분** 이라고 생각하시면 됩니다. 즉, **포인터 배열은 정말로 배열이고, 배열 포인터는 정말로 포인터 였죠.**

```c
/* 포인터배열*/
#include <stdio.h>
int main() {
  int *arr[3];
  int a = 1, b = 2, c = 3;
  arr[0] = &a;
  arr[1] = &b;
  arr[2] = &c;

  printf("a : %d, *arr[0] : %d \n", a, *arr[0]);
  printf("b : %d, *arr[1] : %d \n", b, *arr[1]);
  printf("b : %d, *arr[2] : %d \n", c, *arr[2]);

  printf("&a : %p, arr[0] : %p \n", &a, arr[0]);
  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
a : 1, *arr[0] : 1
b : 2, *arr[1] : 2
b : 3, *arr[2] : 3
&a : 0x7ffe8a2fa4e4, arr[0] : 0x7ffe8a2fa4e4
```

마지막 출력결과는 여러분과 상이할 수 있으나 두 값이 같음을 주목하세요.

일단, `arr` 배열의 정의 부분을 봐봅시다.

```c
int *arr[3];
```

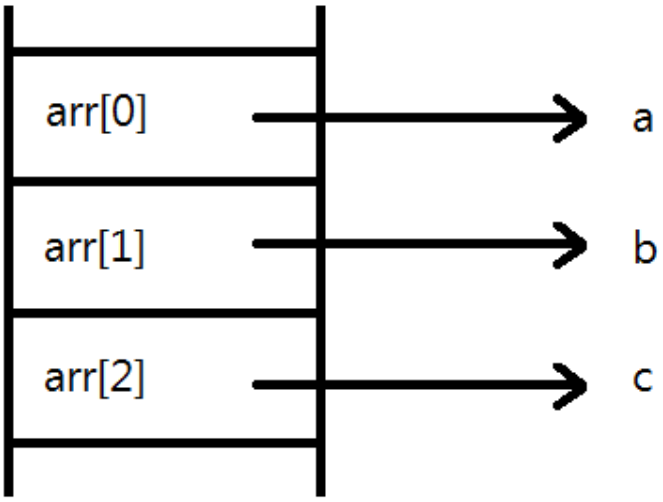
위 정의가 마음에 와닿나요? 사실, 저는 처음에 배울 때 별로 와닿지 않았습니다. 사실, 이전에도 말했듯이 위 정의는 아래의 정의와 동일합니다.

```c
int* arr[3];
```

이제, 이해가 되시는지요? 우리가 배열의 형을 `int, char` 등등으로 하듯이, 배열의 형을 역시 `int*` 으로도 할 수 있습니다. 다시말해, 배열의 각각의 원소는 *int* 를 가리키는 포인터 형으로 선언된 것입니다. 따라서, `int` 배열에서 각각의 원소를 `int` 형 변수로 취급했던 것처럼 `int*` 배열에서 각각의 원소를 포인터로 취급할 수 있습니다. 마치, 아래처럼 말이지요.

```
arr[0] = &a;
arr[1] = &b;
arr[2] = &c;
```

각각의 원소는 각각 int 형 변수 a,b,c 를 가리키게 됩니다.이를 그림으로 표현하면 아래와 같습니다.



arr[0] 에는 변수 a 의 주소가, arr[1] 에는 변수 b 의 주소, arr[2] 에는 변수 c 의 주소가 각각 들어가게 됩니다. 이는 마지막 printf 문장에서도 출력된 결과로 확인 할 수 있습니다.

사실, 포인터 배열에 관한 내용은 짧게 끝냈습니다. 하지만, C 언어에서 상당히 중요하게 다루어지는 개념입니다. 아직 여러분이 그 부분에 대해 이야기할 단계가 되지 않았다고 보아, 기본적인 개념만 알려 드린 것입니다. 꼭 잊지 마시길 바랍니다.

자. 이제 배열을 향한 대장정이 끝이 났습니다. 여기까지 부담없이 이해하셨다면 여러분은 C 언어의 성지를 넘게 된 것입니다! 사실, 여러분은 이 포인터를 무려 3 강의를 연달아 들으면서 '도대체 이걸 왜하냐?' 라는 생각이 머리속에 끝없이 멤돌았을 것입니다. 물론, 앞에서도 이야기 했지만 포인터는 다음 단계에서 배울 내용에 필수적인 존재입니다. 사실, 지금은 아무짝에도 쓸모 없는 것 같지만...

여기까지 스크롤을 내리면서도 마음 한 구석에 응어리가 있는 분들은 과감하게 포인터 강좌를 처음부터 읽어 보세요. 저의 경우 포인터만 책 수십권을 찾아보고 인터넷에서 수십개의 자료를 찾아가며 익혔습니다. 그래도 궁금한 내용들은 꼬오옥 댓글을 달아주세요. 저는 정말 아무리 이상하고 괴상한 질문도 환영하니.. 꼭 궁금한 내용을 물어봐주세요 :)

생각 해 볼 문제

문제 1

3 차원 배열의, 배열이름과 동일한 포인터는 어떻게 정의될 것인가? (난이도 : 中)(참조 : 2 차원 배열에선 `int (*arr)[4];` 와 같은 꼴이었다)

문제 2

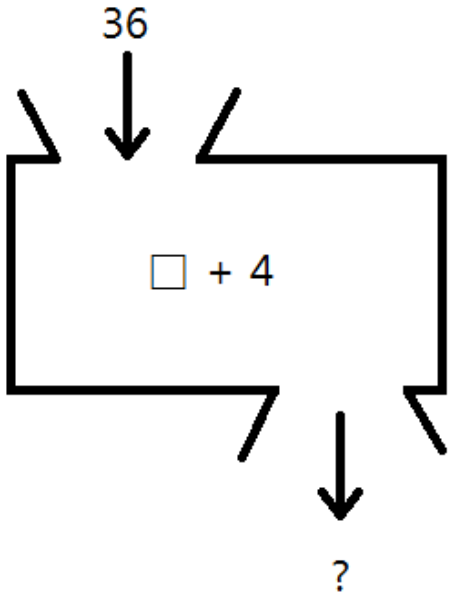
포인터 간의 형변환은 무엇을 의미하는가? 그리고, C 언어에서 포인터 간의 형변환이 위험한 것인가? (난이도 : 中)(참고적으로, 포인터간의 형 변환은 아직 이야기 한 적이 없으나 한 번 시도는 해보세요)

11

함수 (function)

안녕하세요 여러분. 이 강좌를 읽고 있을 여러분은 포인터의 고지를 정복하고 온 위대한 전사(?)들 입니다. 이제, 앞으로 다룰 내용은 C 언어에서 중요하면서도 쉬운 부분 이니 큰 부담 없이 편히 읽으시면 합니다. 또한, 앞에서 배운 포인터를 이제 본격적으로 활용하는 단계에 접어들기 때문에 혹여라도 잊은 것이 있는지 없는지 매일 한 번씩 다시 정독하시면 좋습니다. 또는, 다른 C 언어 강좌로 한 번 더 공부해 보세요. 다른 방식으로 공부하다 보면 이해가 더 잘될 수 도 있습니다.

저는 제 강좌에서 여러분이 최소한 초등학교 4 학년 정도의 수학을 이수하셨다면 아래와 같은 문제를 본 기억이 어렴풋이나마 있을 것 입니다.



음.. 아래 물음표에서 어떤 값이 출력될까요? 아마, 여러분 대부분은 '40' 이라고 짐작하실 것입니다. 맞습니다. 40 입니다. 위 마술 상자는 입력 받은 값에 4 를 더해서 출력하는 상자 입니다. 만일 우리가 36 이 아니라 10 을 집어넣었다면 14 가 나왔을 것이지요.

수학에서 함수는 마술 상자와 비슷합니다. 특정한 값을 입력 받아, 이 값을 가지고 상자 내부에서 지지고 볶고 해서 결과를 내보낸 마술 상자 처럼, 수학에서는 특별한 값 x 를 입력 받아 지지고 볶은 뒤(유식한 말로 연산을 취하여)에 결과를 출력하는 것을 함수 라고 합니다. (참고적으로, 수학에서 보통 입력값은 x, 출력값은 y 라고 하니, 아래에선 아무런 이야기 없이 사용하도록 하겠습니다.)

수학에서 마술 상자를 글로 표현하기 조금 껄끄러우니 보통 다음의 표현을 사용합니다.

$$y = f(x)$$

이는, 'x 라는 값을 f 라는 마술 상자 (함수) 를 통과시켰더니 y 라는 값이 되었다' 라는 의미와 일맥상통합니다. 위의 마술 상자의 경우 입력값에 4 를 더한 값을 반환하였습니다. 그렇다면, 위의 마술 상자는 아래와 같은 식으로 나타낼 수 있습니다.

$$f(x) = x + 4$$

예를 들어 x 에 36 이 들어간다면 f(x) 의 값, 즉 y 의 값은 $36 + 4$ 인 40 이 됩니다. 따라서, 40 이 출력된다는 사실을 볼 수 있습니다. 그렇다면, 아래의 예를 보고 어떠한 값이 출력되는지 맞추어 보세요.

$$f(x) = x^3 + 2x, f(x) = ?$$

$$g(x) = x^2 - 3x + 4, g(5) = ?$$

(이례적으로 답을 올리자면 $f(3) = 33$, $g(5) = 14$)

간혹 제 블로그를 방문하는 분들 중에는 초등학생인 분들이 있기에, 함수를 전혀 들어보지 못한 분들이 있을 까봐 짤막하게 함수에 대해 설명하였습니다. C 언어의 함수도 비슷한 개념으로 사용됩니다.

함수의 시작

우리가 프로그래밍을 하면서 여러가지 작업들을 반복적으로 해야되는 경우가 종종 있습니다. 예를 들어서 변수 a 와 b 중 최대값을 구하는 것을 생각해봅시다. 우리가 이를 프로그래밍 시에 필요로 하게 된다면 다음과 같이 해야 될 것입니다.

```
int max;
if (a >= b) {
  max = a;
} else {
```

```
  max = b;
}
```

뭐, 위 코드는 아주 아주 쉬운 코드 이니 설명은 하지 않겠습니다. 그런데, 실제로 프로그래밍을 하다 보면 어떠한 두 변수의 최대값을 구하는 경우가 자주 생긴다는 것입니다. 현재 까지 배운 바로는 이러한 상황에서는 코드 복사 붙여넣기를 통해 소스를 채워나가면 된다고 생각했습니다. 자, 그렇다면 이러한 방법이 합리적인 것일까요?

만일 최대값을 구하는 것이 프로그램에서 100 번 정도 필요하다면 그 때 마다 위 코드를 복사해서 변수 이름만 살짝 바꿔주면 됩니다. 하지만, 소스가 얼마나 지저분해질까요? 소스가 수천줄이 넘어가면 위 코드가 무슨 작업을 하는지 눈에 팍 들어오기 힘듭니다.

그렇다면 여러분은 이렇게 생각해 볼 수 있습니다.

> "최대값을 출력하는 함수를 만들어버리자!!"

응? 도대체 위 말이 무슨뜻인감.. 아마도 여러분은 갈피를 잡기 힘들 것입니다. 하지만 이렇게 생각하면 편합니다. **아까 위에서 설명한 마술 상자 처럼 우리가 만들게 될 마술상자는 두 개의 값이 입력된다면 큰 놈을 출력하는 것이야!**

오오. 괜찮은 아이디어 아닌가요. 우리는 그 긴 코드(사실 그렇게 긴 것은 아니지만;;) 를 매번 쓰는 대신에 두 값을 입력받아서 큰 것을 출력하는 마술 상자 (함수) 를 제작하여, 최대값을 구하는 것이 필요할 때 마다 그 마술 상자에 두 변수를 넣어 버리면 되지 않습니까? 그러면 우리는 그 마술 상자가 뱉어내는 값을 받아 먹기만 하면 되는 것이니까요.

자, 그럼 마술 상자를 만들어봅시다~

일단, 최대값을 구하는 함수를 만들어 보기 전에 아주 아주 간단한 함수를 먼저 만들어보겠습니다.

```c
#include <stdio.h>
/* 보통 C 언어에서, 좋은 함수의 이름은 그 함수가
무슨 작업을 하는지 명확히 하는 것이다. 수학에서는
f(x), g(x) 로 막 정하지만, C 언어에서는 그 함수가 하는
작업을 설명해주는 이름을 정하는 것이 좋다. */
int print_hello() {
  printf("Hello!! \n");
  return 0;
}
int main() {
  printf("함수를 불러보자 : ");
  print_hello();

  printf("또 부를까? ");
  print_hello();
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
함수를 불러보자 : Hello!!
또 부를까? Hello!!
```

음.. 일단 우리가 여태까지 보아왔던 것과 매우 다른 모습입니다. 하지만 걱정하지 마세요. 금세 이해하게 될 것이니까요.

```
int print_hello() {
  // 잠시 생략
}
```

일단, 함수의 **정의(definition)** 부분을 살펴 봅시다. 위와 같이 `int print_hello()` 라고 써 있는 부분을 함수의 '정의' 부분이라 부릅니다. 우리는 함수의 정의 부분에서 3 가지 사실을 알 수 있는데, 일단은 2 가지만 먼저 설명하고 나머지 하나는 아래에서 설명하겠습니다.

먼저, 우리에게 친근한 키워드가 하나 있습니다. 바로 `int` ! 우리는 여태까지 `int` 를 변수나 배열을 정의하는데만 사용하였습니다. 그런데, `int` 가 놀랍게도 함수를 정의하는데도 사용되고 있습니다. 여기서의 `int` 는 다음과 같은 사실을 알려줍니다. '이 함수는 `int` 형의 정보를 반환한단다~'. 반환? 그렇다면 반환은 또 뭐야.

우리가 앞서, 마술 상자를 이야기 하였을 때, 우리가 36 을 마술 상자에 넣는다면, 4 를 더해서 40 을 출력한다고 하였습니다. 이 때, 우리는 '출력된다' 라는 사실을 함수에서는 '반환한다' 라고 이야기 합니다. 영어로는 `return` 이라고 하지요.

```
int print_hello() {
  printf("Hello!! \n");
  return 0;
}
```

위 함수 정의 부분에서, 아래에서 두번째 줄에 `return 0;` 라고 써있는 부분을 볼 수 있습니다. 이 함수는 0 을 반환한다는 뜻이군요. 즉, 우리가 위와 같은 마술 상자를 이용한다면 언제나 0 이 출력됩니다. 이 때, 함수의 반환형이 `int` 이므로, 0 은 `int` 의 형태로 저장되어 나갑니다.

여기서 `int` 의 형태로 저장된다는 말의 의미는 0 이라는 데이터가 메모리 상의 4 바이트를 차지하여 반환된다는 뜻이지요. (통상적으로 정수를 반환하는 함수들은 모두 `int` 를 사용합니다.)

함수의 정의 부분에서 알 수 있는 두 번째 사실은 바로 함수의 이름 입니다. 대충 짐작이 가듯이, 위 함수의 이름은 `print_hello` 입니다. 끝에 붙는 `()` 는 함수의 이름에 포함되는 것이 아닙니다. 끝에 붙는 괄호 두 개는 이것이 함수라는 사실을 의미합니다. 만일 우리가 끝에 `()` 를 붙이지 않는

다면 `int print_hello` 라는 문장은 단순히 끝에 `;` 를 제대로 붙이지 않았구나 라고 해석되어 오류를 출력하게 됩니다. 꼭 `()` 를 붙여주세요~

주석에서도 잘 설명 하였듯이 좋은 함수 이름의 조건은 함수가 무슨 일을 하는지에 대해서 잘 설명하는 것 입니다. 만일 우리가 함수를 `int asdfasd()` 라고 만들었다면 우리가 `asdfasd` 라는 함수를 보고 무슨 일을 하는지 잘 알 수 없습니다.

하지만 우리의 예제 처럼 `print_hello` 라고 하게 된다면 이 함수가 대략 'hello 를 출력하는구나' 라는 사실을 알 수 있겠지요. 다만, 함수의 이름이 너무 길어지면 함수를 사용시 너무 불편하므로 20 자가 넘어가게 하지는 맙시다. 또한, 함수의 이름 역시 변수의 이름 조건과 동일하므로 기억나지 않는 분들은 3강 변수가 뭐지? 의 맨 마지막 부분을 보세요.

```
int print_hello() {
  printf("Hello!! \n");
  return 0;
}
```

함수의 정의부분은 그만 살펴보고, 이제 함수가 무슨 일을 하는지 알 수 있는 부분을 살펴 봅시다. 이 부분은 보통 함수의 **몸체(body)** 라고 부릅니다. 이번 예제 함수의 몸체는 설명을 안해도 잘 알 수 있습니다. 이 함수는 `printf("Hello!! \n");` 을 실행한 후, 0 을 반환한다 이지요?

```
printf("함수를 불러보자 : ");
print_hello();

printf("또 부를까? ");
print_hello();
```

마지막으로 실제로 함수를 호출하는 부분을 살펴 봅시다. 함수를 불러내는 방법(보통 **호출한다(call)** 라는 표현을 사용하므로 앞으로 호출한다고 표현하겠습니다) 은 단순히 함수의 이름을 써주시기만 하면 됩니다. 물론 그 뒤에 `()` 도 붙여주어야 겠지요.

다시 말하지만 `()` 는 함수의 이름에 포함되는 것이 아닙니다. 하지만, `()` 를 써줌으로써 컴파일러에게 '내가 지금 쓴 것이 함수 이니라~' 라는 사실을 말해주게 되는 것이지요. 만일 함수를 호출한답시고 `print_hello;` 라고 쓴다면 컴파일러는 '어딘가에 `print_hello` 라는 변수에 접근하였네' 라고 생각하는데, `print_hello` 라는 변수가 없으므로 오류를 출력하게 됩니다.

함수를 호출하면 프로그램은 함수의 내용을 실행하게 됩니다. 그리고 다시, 원래 실행되려는 부분으로 돌아오게 되죠. 위의 경우, "함수를 불러보자" 가 출력된 후 `print_hello()` 를 통해 함수를 호출하였습니다. 그러면 프로그램은 `print_hello()` 라는 함수로 넘어가서, 이 함수의 내용을 다 실행한 뒤에 다시 원래 있던 곳으로 돌아와 넘어가게 됩니다.

이와 같은 현상은 실생활에서도 볼 수 있습니다. 밥을 먹고 있다가 '엄마가 부르신다' 라는 함수가 호출되면 엄마한테로 달려갑니다. 그리고 '엄마가 부르신다' 라는 함수가 종료되면 다시 밥 먹던

식탁으로 와서 밥을 먹게 되지요. 이 때, 함수의 종료는 두 가지 형태로 있을 수 있습니다. 하나는 반환이 되어 종료를 하게 되는 것이고 다른 하나는 함수의 끝 부분 까지 실행하여 종료되는 것입니다. 함수는 반환을 하여 종료되는 것이 안전합니다. 한 가지 중요한 사실은 return 을 실행하면 함수는 무조건 종료되어 함수를 호출하였던 부분을 돌아간다는 점입니다.

```c
/* 함수의 리턴 */
#include <stdio.h>
int return_func() {
  printf("난 실행된다 \n");
  return 0;
  printf("난 안돼 ㅠㅠ \n");
}
int main() {
  return_func();
  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
난 실행된다
```

물론 앞에서 이야기 하였듯이 짐작은 하고 있으셨겠지만 확실히 보여드리기 위해 예제를 작성하였습니다.

```c
int return_func()
```

연습 삼아 위 부분이 무슨 의미인지 다시 한 번 살펴봅시다. 일단, int 를 보아 이 함수는 int 형을 리턴한다는 의미이고, return_func 을 보아서 이 함수의 이름이 return_func 라는 사실을 알 수 있습니다.

```c
{
  printf("난 실행된다 \n");
  return 0;
  printf("난 안돼 ㅠㅠ \n");
}
```

다음은 함수의 몸체 입니다. 앞에서 이야기 하였듯이 return 이 실행되면 프로그램은 바로 함수를 호출하였던 부분으로 넘어가 버려 그 다음에 오는 모든 것들(위 예제에선 printf("난 안돼 ㅠㅠ \n");) 이 실행되지 않게 됩니다.

```c
/* 반환값 */
#include <stdio.h>
int ret() { return 1000; }
int main() {
  int a = ret();
  printf("ret() 함수의 반환값 : %d \n", a);

  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
ret() 함수의 반환값 : 1000
```

마지막으로 한 번더, 함수의 정의 부분을 분석해봅시다.

```c
int ret()
```

아마 이쯤 되면 여러분은 위 것만 보고도 이 함수는 이름이 `ret` 이고, `int` 형을 반환한다 라는 사실을 알 수 있을 것 입니다.

그리고 `ret` 함수의 몸체를 살펴 보자면 상당히 간단하다라는 것을 알 수 있습니다.

```c
{ return 1000; }
```

그리고 위 코드는 "이 함수를 호출하면 1000 을 리턴한다" 정도 되겠지요.

```c
int main() {
  int a = ret();
  printf("ret() 함수의 반환값 : %d \n", a);

  return 0;
}
```

위는 `ret()` 함수를 호출하여 그 값을 `a` 에 대입하는 문장 입니다. 그런데 `ret()` 가 가지는 값이 있나요? 물론, `ret()` 는 함수이기 때문에 위와 같이 이용하면 안될것 같습니다만, `ret()` 를 코드에 쓰게 된다면 이 말은 "`ret()` 함수의 반환값" 라는 의미를 가집게 됩니다. 즉, 컴퓨터가 위 코드를 실행한다면 `a` 에는 `ret` 함수의 반환값인 1000 이라는 값이 들어가게 됩니다.

아무튼 아래의 유명한 격언을 기억하시기 바랍니다.

호랑이는 죽어서 가죽을 남기고, 함수는 죽어서 리턴값을 남긴다!

메인(main) 함수

아마 꼼꼼하신 여러분들은 이미 `int main()` 이란 부분도 `main` 이라는 함수를 정의하고 있다는 사실을 눈치 채고 있을 것입니다. 맞습니다. 여러분은 `main` 이라는 이름의 함수를 정의하고 있는 것이였습니다. 그런데 왜 하필이면 `main` 일까요?

왜냐하면 프로그램을 실행할 때 컴퓨터가 `main` 함수 부터 찾기 때문입니다 (물론 모든 경우가 그런 것은 아니고 적어도 우리가 앞으로 만들게 될 C 프로그램들의 경우). 즉, 컴퓨터는 프로그램을 실행할 때 프로그램의 `main` 함수를 호출함으로써 시작합니다. 만일 `main` 함수가 없다면 컴퓨터는 프로그램의 어디서 부터 실행할 지 모르게 되어 오류가 나게 되죠.

보통 메인 함수를 아래와 같은 형태로 정의합니다.

```
int main()
```

위에서 배운 내용을 살짝 활용하면 "이 함수는 리턴형이 `int` 이고 이름은 `main` 이네!" 정도 알 수 있겠지요. 그런데, 메인 함수가 리턴을 하면 누가 받을까요? 메인 함수가 프로그램 맨 처음에 실행되는 함수라면, 맨 마지막으로 종료되는 함수도 메인 함수 이기 때문에 리턴값을 받을 수 있는 함수가 없을 듯 합니다.

사실, 그렇지 않습니다. 메인 함수가 리턴하는 데이터는 바로 운영체제가 받아들입니다. 운영체제. 즉 여러분이 아마도 쓰고 계실 Windows XP 나 Linux 에서 받는 다는 이야기 이지요.[1] 보통 메인 함수가 정상적으로 종료되면 0 을 리턴하고, 비정상적으로 종료되면 1 을 리턴한다고 규정되어 있습니다. 우리가 여태까지 만들어왔던 모든 메인 함수들은 정상적으로 종료되므로 마지막에 0 을 리턴하였죠. 사실, 1 을 리턴한다고 해서 큰 문제는 없습니다. 이 정보를 활용하는 경우는 매우 드물기 때문이죠.

아무튼, 여기서 알아야 할 사실은 "`main` 도 함수다!" 정도만 알아 두셨으면 합니다.

이번에는 맨 위에서 구상하였던 마술 상자 (4 를 더한값을 출력하는..) 를 제작해보기로 하였습니다. 일단 여러분은 아래와 같이 구현할 수 있지 않을까 라는 것을 머리속에 떠올릴 것입니다.

```
/* 마술 상자 */
#include <stdio.h>
int magicbox() {
  i += 4;
  return 0;
}
int main() {
```

1) 이 강좌는 2009 년에 나왔습니다.

```
  int i;
  printf("마술 상자에 집어넣을 값 : ");
  scanf("%d", &i);

  magicbox();
  printf("마술 상자를 지나면 : %d \n", i);
  return 0;
}
```

컴파일 하면 아래와 같이 달콤한 오류를 볼 수 있습니다.

컴파일 오류

```
error C2065: 'i' : 선언되지 않은 식별자입니다.
```

아니, 왜? 이런 오류가 뜨는 것이지.. 분명히 우리는 `main` 함수 내에서 `i` 라는 이름의 `int` 형 변수를 선언하였고 다른 함수(여기선 `magicbox`) 에서 사용할 수 있어야 되는 것 아닌가요? 하지만 안타깝게도 아닙니다. 사실, 이 마술상자는 우리가 생각했던 것 보다도 훨씬 멋진 개념입니다.

어떠한 함수를 호출할 때, 호출된 함수는 함수를 호출한 놈에 대해서 어떠한 것도 알고 있지 않습니다. 즉, 내가 `magicbox` 라는 함수를 호출하였을 때, 이 `magicbox` 는 내가 얘를 호출하였는지, 다른 애가 (즉, 다른 코드를 말하는 것이겠죠;;) 얘를 호출하였는지 '전혀 알 수 없다' 라는 것입니다.

```
int magicbox() {
  i += 4;
  return 0;
}
```

따라서 이 함수는 `i` 라는 변수에 대해서 아무런 정보도 가지지 않고 있습니다. 왜냐하면 이 함수를 호출한 것이 무엇인지에 대한 정보가 하나도 없기 때문이죠. 결과적으로 `main` 함수에서 정의된 `i` 라는 변수는 `magicbox` 의 입장에서 본다면 듣도 보도 못한 것이 되는 것입니다. 결과적으로 위에서 보았던 오류와 같이 `i` 라는 변수가 선언되어있지 않다는 오류를 내게 됩니다.

아직도 위 코드가 왜 작동이 되지 않는지 이해가 되지 않으신 분들은 아래의 옛날 이야기(?) 를 보시면 됩니다.

옛날 옛날 이집트 시대에 어떤 부유한 귀족이 있었습니다. 이 귀족은 하루에 10000 달러씩 장사를 해서 벌었습니다. 그런데 공교롭게도 수학을 매우매우 못했죠. 따라서, 이 귀족은 노예를 한 명 사서, 이 노예에게 자신의 현재 재산에 10000 을 더해서 알려 달라고 하였습니다. 그리고 시간이 흘러 10 시간 뒤, 귀족의 일과가 끝났습니다. 이제, 그는 오늘 자신의 재산 현황을 파악하기 위해서 노예를 호출했습니다.

야 말해

그런데 노예는 아무 말도 하지 못했습니다.

> 야 말하라고, 내 재산에 10000 을 더해서 말하라니까

역시 아무말도 없었습니다. 왜일까요? 그야, 당연히 노예는 귀족의 재산에 대한 정보가 없었기 때문입니다. 귀족이 방금 노예를 호출함으로써 한 일은, "자신의 재산 `+= 10000`" 이였습니다. 그런데, '자신의 재산' 이란 변수는 노예의 머리에서 정의된 것이 아니므로 알 노릇이 없습니다.

그렇다면, 이제 아무 쓸모 없게된 불쌍한 노예를 악랄한 귀족이 죽이게 내버려 두어야 하나요? 물론, 그리하면 안되겠죠. 일단, 여기서 문제점을 해결하기 위해선 노예가 "현재 귀족의 재산" 이라는 데이터만 머리에 넣고 있으면 됩니다. (노예가 계산을 충분히 잘한다는 가정 하에..) 이 말을, C 언어 적으로 이야기 하면 노예라는 함수에 "주인의 현재 재산" 이라는 변수를 정의하고 이 변수에 "자신(주인)의 재산" 의 값을 넣은 뒤에, "주인의 현재 재산+=10000" 을 계산한 후, "주인의 현재 재산" 을 반환(입으로 말함) 하면 되는 것입니다.

이제, 문제는 노예 머리속에 "주인의 현재 재산" 이라는 변수에 "자신(주인) 의 재산" 값을 어떻게 넣느냐가 문제 입니다. 바로 아래에서 보도록 하죠.

함수의 인자

```c
#include <stdio.h>
int slave(int master_money) {
  master_money += 10000;
  return master_money;
}
int main() {
  int my_money = 100000;
  printf("2009.12.12 재산 : $%d \n", slave(my_money));

  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
2009.12.12 재산 : $110000
```

일단, 함수의 정의 부분이 바뀐 것을 볼 수 있습니다.

```c
int slave(int master_money)
```

slave 가 함수 임을 알려주는 소괄호 안에 int master_money 가 써 있군요. 이는 다음과 같은 의미를 가집니다.

"나를 호출하는 코드로 부터 어떤 값을 mater_money 라는 int 형 변수에 인자(혹은 매개변수라고도 부름)로 받아들이겠다!"

허걱.. 정말 뭔소린지 알 수 없군요. 먼저 '인자' 가 무엇인지 살펴 보도록 합시다. 아까 전에 우리는 노예의 머리속에 '현재 주인이 가지고 있는 재산' 이라는 값을 어떻게 입력해야 할지가 문제라고 하였습니다. 그런데, slave 함수와 main 함수는 전혀 별개의 함수 이기 때문에 slave 함수는 main 함수 안의 변수를 사용할 수 없을 뿐더러 main 함수에서도 slave 함수의 변수들이 무엇인지 전혀 알 길이 없습니다.

하지만, **인자(argument, 혹은 매개변수(parameter) 라고 부른다)** 를 이용하면 이러한 일을 가능하게 합니다. 일단, 인자는 직관적으로 봐도 알 수 있듯이 slave 함수 내에 선언이 되어 있는 변수 입니다. 이 때, 인자는 함수 정의할 때의 소괄호 안에 나타나게 되죠. 위의 경우 slave 함수는 int 형의 master_money 라는 변수를 인자로 가지고 있습니다. 이제, 이 함수를 어떠한 함수에서 호출을 한다고 합시다. 그렇다면, 이 함수를 호출 할 때, 인자에 적당한 값을 넣어 주어야 합니다. 마치 아래와 같이요.

```
slave(500);
```

이 말은 **slave 함수를 호출할 때, slave 함수 안에서 정의된 master_money 라는 변수에 500 이라는값을 전달하겠다!** 라는 의미입니다. 따라서, slave 함수 내부에 정의된 master_money 라는 변수에는 500 이라는 값이 들어가게 됩니다. 그렇다면 아래는 어떨까요?

```
slave(my_money);
```

이 것도 마찬가지 입니다. 이렇게 이용한다면 "slave 함수를 호출할 때, slave 함수 안에서 정의된 master_money 라는 변수에 my_money 의값을 전달하겠다!" 가 되겠지요. 만일 my_money 에 10000 이 있었더라면 slave 함수를 호출 시에 master_money 에는 10000 이 들어가게 됩니다. 결론적으로 말하자면 함수의 인자는 '함수를 호출한 것과, 함수를 서로 연결해 주는 통신 수단' 이라고 말할 수 있습니다. 이러한 연유에서 수학적인 용어로 틀린 표현 이지만 C 에선 '매개 변수' 라고 부릅니다.

그렇다면, 위의 예제를 한 번 살펴볼까요?

```
int main() {
  int my_money = 100000;
  printf("2009.12.12 재산 : $%d \n", slave(my_money));

  return 0;
}
```

일단, slave 함수를 호출하는 **호출자(caller)** 의 코드를 살펴봅시다. printf 에서, 맨 뒤에 %d 에 들어갈 값으로 slave(my_money) 가 반환 하는 값을 넣었습니다. slave(my_money) 가 반환하는 값을 먼저 넣기 위해선 slave 함수를 호출해야 하는데 이 때 my_money 의 값이 slave 함수의 인자로 전달이 됩니다. 그러면 slave 함수는 아래의 코드를 실행하겠지요.

```
{
  master_money += 10000;
  return master_money;
}
```

즉, master_money 에 10000 을 더한 후, 그 값을 반환하게 됩니다. 따라서, 100000 에 10000 이 더해진 110000 이 출력되겠지요.

이번에는 과연 성공적으로 컴파일 될지 의문이 드는 예제를 한 번 만들어 보았습니다.

```
/* 될까용 */
#include <stdio.h>
int slave(int my_money) {
  my_money += 10000;
  return my_money;
}
int main() {
  int my_money = 100000;
  printf("2009.12.12 재산 : $%d \n", slave(my_money));
  printf("my_money : %d", my_money);

  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
2009.12.12 재산 : $110000
my_money : 100000
```

아마도, 앞의 내용을 열심히 배우신 분들은 위 코드가 정상적으로 실행될 것이라는 것을 알고 계셨겠죠? 하지만, 그렇지 못한 분들을 위해 설명 하자면

```
int slave(int my_money) {
  my_money += 10000;
  return my_money;
}
```

위 slave 함수는 my_money 를 인자로 받고 있습니다. 여기서 중요한 점은 my_money 가 slave 의 변수라는 것입니다. 그렇다면 slave 함수를 호출하는 부분을 볼까요.

```
int main() {
  int my_money = 100000;
  printf("2009.12.12 재산 : $%d \n", slave(my_money));
  printf("my_money : %d", my_money);

  return 0;
}
```

음, slave 함수를 호출할 때 main 함수 내부에서 선언된 my_money의 값을 slave 함수의 변수인 my_money 에 전달하고 있습니다. 즉, 각 함수 내부에서 선언된 my_money 들은 이름은 같지만 **서로 다른 변수** 이고, 메모리 상의 **다른 위치**를 점유하고 있습니다. 즉, 우리가 보기에 두 변수는 똑같은 것으로 보여도 적어도 컴퓨터가 보기에는 두 변수는 서로 다른 것들입니다.

두 번째로 주목할 점은 **값**이 전달된다는 것입니다. 이는 아까 제가 위에서 부터 누누히 강조해 온 점이기도 한데, slave 함수를 호출할 때 slave 함수의 my_money 인자에는 값이 전달됩니다. 즉, main 함수의 my_money 의 100000 이라는 값이 slave 함수의 my_money 라는 인자에 저장되어 들어갑니다.

따라서, slave 함수에서 my_money의 값을 아무리 지지고 볶아도 main 함수의 my_money 변수에는 전혀 영향을 주지 않는다는 것이지요. 왜냐하면 slave 함수의 my_money 변수는 단지 **main 함수의 my_money 와 같은 값을 가진 채로 초기화된 메모리 상의 또다른 변수** 이기 때문이지요. 이건 마치

```
int a = b; b++;
```

이라고 했는데 a 의 값이 b 와 같이 1 증가함을 바라는 것과 같습니다. 아무튼, 결과적으로 main 함수에서 두 번째 printf 문에서 main 함수의 my_money 의 값을 출력했을 때 에는 전혀 변하지 않은 100000 이 출력됩니다.

그렇다면 우리가 다른 함수의 변수의 값을 수정하고자 하는 함수를 만들고 싶다면 어떻게 해야 될까요? 우리가 앞에서 배운 내용을 생각해보면 "각 함수의 세계는 너무나 배타적이여서 각 함수는 서로에 무슨 변수가 있는지 모른다. 사실 (정확히 말하자면 각 함수의 형태(리턴형, 함수의 이름, 인자들의 형(type)) 빼고는) 서로에 대해 아는 것이 완전히 없다."

그럼, 정말로 우리는 다른 함수에서 정의된 변수의 값을 수정하는 함수는 결코 작성할 수 없는 것일까요?

답은 아니오 입니다. 놀랍게도 포인터를 이용하면 됩니다 (드디어 포인터가 쓸모 있어지나요?). 일단, 이것까지 이야기 하면 강좌가 너무 길어지므로 오늘은 이쯤에서 끝내도록 하고 어떻게 포인터로 가능할까 에 대해서 다음 강좌가 나올 때 까지 생각해봅세요.

생각해보기

문제 1

이 강좌 최상단에서 이야기 했던 마술 상자를 함수로 제작해보세요 (난이도 : 못한다면 강좌를 다시 읽어보아야 할 것입니다)

문제 2

어느날 귀족이 돈벌이가 시원치 않아져서 이전에는 일정하게 10000 달러씩 챙겼지만 이제 일정치 않은 수입을 얻게 되었습니다. 여러분은 `slave` 함수를 인자를 2 개를 가져서, 하나는 현재 귀족의 재산, 다른 하나는 오늘 귀족의 수입을 인자로 전달받는 새로운 함수를 만들어 보세요 (난이도 : 下)

문제 3

1 부터 `n` 까지의 합을 구하는 함수를 작성해보세요. 수학적인 공식을 써도 되지만 `for` 문으로 작성하는 것이 연습 하는데에는 도움이 될듯 합니다. (난이도 : 下 1 부터 `n` 까지의 합을 구하는 함수를 작성해보세요. 수학적인 공식을 써도 되지만 `for` 문으로 작성하는 것이 연습 하는데에는 도움이 될듯 합니다. (난이도 : 下)

문제 4

`N` 값을 입력 받아서 1 부터 `N` 까지의 소수의 개수를 출력하는 함수를 제작해보세요. (난이도 : 下)

문제 5

특정한 수 `N` 을 입력받아서 `N` 을 소인수분해한 결과가 출력되게 해보세요 (난이도 : 中)

예) `factorize(10);` 출력결과 : 2 Œ 5

`factorize(180);` 출력결과 : 2 Œ 2 Œ 3 Œ 3 Œ 5

문제 6

`int function(int *arg)` 와 같은 함수가 무엇을 뜻하는지 생각해보세요

포인터로 받는 인자

안녕하세요 여러분. 이전에 함수에 대해선 잘 이해하셨는지요? 그리고, 마지막에 던진 의미 심장한(?) 질문에는 답을 구하셨나요? 우리는 이전 12 강에서 포인터에 대해서 다루어왔습니다. 그 때 동안 늘 머리속에 맴돌았던 생각은 "도대체 이거 어따가 써먹는거야?" 였죠. 하지만, 이번 강좌에서 그 질문에 대한 해답을 찾을 수 있기 바랍니다.

일단, 간단히 이전에 포인터에 대해서 배웠던 내용을 리뷰 하자면

포인터는 특정한 변수의 메모리 상의 주소값을 저장하는 변수로, `int` 형 변수의 주소값을 저장하면 `int*`, `char` 이면 `char*` 형태로 선언된다. 또한 `*` 단항 연산자를 이용하여, 자신이 가리키는 변수를 지칭할 수 있으며 `&` 연산자를 이용하여 특정한 변수의 주소값을 알아낼 수 있다.

만일 위 내용중에 한 마디라도 이해가 안되는 부분이 있다면 12 강 포인터 강좌를 다시 읽어 보시기를 강력하게 권합니다. 그렇지 않다면 아래의 내용을 계속 읽어가도록 하죠. 우리는 지난 강좌에서 다음과 같이 단순한 형태로는 다른 함수에서 정의된 변수의 값을 바꿀 수 없다고 했습니다.

```c
/* 이상한 짓 */
#include <stdio.h>
int change_val(int i) {
  i = 3;
  return 0;
}
int main() {
  int i = 0;

  printf("호출 이전 i 의 값 : %d \n", i);
  change_val(i);
  printf("호출 이후 i 의 값 : %d \n", i);

  return 0;
}
```

왜 `main` 함수 안에서 정의된 `i` 의 값이 바뀌지 않는지는 잘 아시겠지만 그래도 한 번 확인해봅시다.

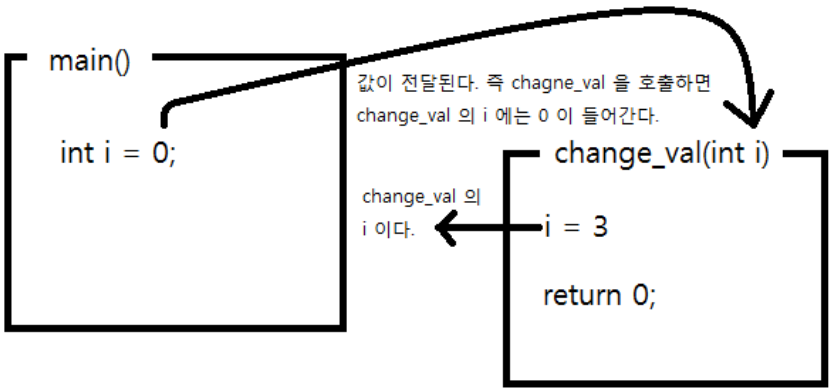
실행 결과

```
호출 이전 i 의 값 : 0
호출 이후 i 의 값 : 0
```

`i` 의 값이 전혀 바뀌지 않았음을 알 수 있습니다. 그 이유는 함수 `change_val` 을 호출 할 때, `change_val` 함수 안에서 정의된 변수 `i` 는 `main` 함수의 `i` 의 값을 전달 받은 후에, `change_val` 함수 안에서 정의된 변수 `i` 의 값을 3 으로 변경하게 됩니다.

여기서 중요한 점은 'main 함수의 i 가 아닌 change_val 함수 안에서 정의된 변수 i 의 값이 3 으로 변경' 된다는 것이지요. 결론적으로 main 함수의 i 의 값에는 아무런 영향도 미치지 못하고 위와 같은 현상이 벌어지는 것입니다.

위 과정을 그림으로 표현하면 아래와 같습니다.



하지만, 여러분은 지난 3 개의 강좌를 통해 포인터에 대해 귀가 아플 만큼 들어 보았을 것입니다. 그리고, 여기에서 그 아이디어를 적극적으로 활용하고자 합니다. 이전의 방법을 통해서 다른 함수에 정의된 변수들의 값을 변경할 때 직면했던 문제는 바로 각 함수는 다른 함수의 변수들에 대해 아는 것이 아무것도 없다는 것이였습니다. 즉 A 라는 함수에서 i 라는 변수를 이용한다면 컴파일러는 이 변수 i 가 오직 A 함수에서만 정의되었다고 생각하지 다른 함수에서 정의되었는지는 상관하지 않다는 것입니다.

그렇지만 궁여지책으로 유일하게 가능했던 것은 인자를 이용해서 다른 함수에 정의된 변수들의 '값' 을 전달하는 것이였습니다. 하지만 그렇게 해도 여전히 불가능해 보였습니다.

```c
/* 드디어 써먹는 포인터 */
#include <stdio.h>
int change_val(int *pi) {
  printf("----- chage_val 함수 안에서 -----\n");
  printf("pi 의 값 : %p \n", pi);
  printf("pi 가 가리키는 것의 값 : %d \n", *pi);

  *pi = 3;

  printf("----- change_val 함수 끝~~ -----\n");
  return 0;
}
int main() {
  int i = 0;

  printf("i 변수의 주소값 : %p \n", &i);
  printf("호출 이전 i 의 값 : %d \n", i);
  change_val(&i);
  printf("호출 이후 i 의 값 : %d \n", i);
```

```
  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
i 변수의 주소값 : 0x7ffd3928afc4
호출 이전 i 의 값 : 0
----- chage_val 함수 안에서 -----
pi 의 값 : 0x7ffd3928afc4
pi 가 가리키는 것의 값 : 0
----- change_val 함수 끝~~ -----
호출 이후 i 의 값 : 3
```

여러분의 출력결과와 다를 수 있습니다.

헉! 눈으로 보고도 믿기지 않으십니까? 호출 이후의 i 의 값이 0 에서 3 으로 바뀌었습니다. 이게 무슨일 입니까? 이건 우리가 여태까지 꼭 하고야 말겠던 바로 그 작업 아닙니까. 바로 다른 함수에서 정의된 변수의 값을 바꾸는 것 말이죠. 그런데, 위 코드를 조금씩 뜯어 들여보다 보면 방법은 매우 간단하다는 것을 알 수 있습니다. 물론, 이 강의를 보고 계시는 일부 똑똑한 독자들은 이미 짐작 했을 것이지만요.

```
int change_val(int *pi)
```

일단, 함수의 정의부분을 살펴보자면 int 형의 변수를 가리키는 pi 라는 이름의 포인터로 인자를 받고 있습니다. 그리고 main 함수에서 이 함수를 어떻게 호출했는지 보면

```
change_val(&i);
```

즉, 인자에 main 함수에서 정의된 i 라는 변수의 '주소값' 을 인자로 전달하고 있습니다. 따라서 change_val 함수를 호출하였을 때 pi 에는 i 의 주소값이 들어가게 됩니다. 즉, pi 는 i 를 가리키게 됩니다.

```
{
  printf("----- chage_val 함수 안에서 -----\n");
  printf("pi 의 값 : %p \n", pi);
  printf("pi 가 가리키는 것의 값 : %d \n", *pi);

  *pi = 3;
```

```
  printf("----- change_val 함수 끝~~ -----\n");
  return 0;
}
```

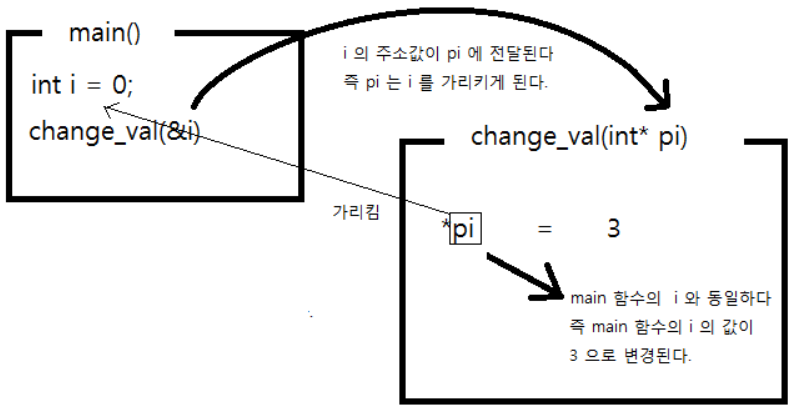
pi 가 i 의 주소값을 가지고 있으므로 pi 를 출력했을 때 그 값은 i 의 주소값과 같을 수 밖에 없습니다. 이는 두 번째 printf 문장에서 확인할 수 있습니다. 또한 그 아래 *pi 를 통해서 i 를 간접적으로 접근할 수 있습니다. 왜냐하면 * 라는 단항 연산자의 의미가 '내가 가지는 주소값에 해당하는 변수를 의미해라' 이기 때문에 *pi 는 pi 가 가리키고 있는 변수인 i 를 의미할 수 있게 됩니다. 즉, pi 를 통해서 굳게 떨어져 있던 main 과 change_val 함수의 세계 사이에 다리가 놓이게 되는 것이지요.

간혹 pi 가 main 함수에서 정의된 것이라고 착각하는 분들이 있는데, pi 역시 change_val 함수 내에서 정의된 변수 입니다.

또한 *pi = 3 을 통해 'pi 가 가리키고 있는 변수' 의 값을 3 으로 변경할 수 있습니다. 여기서 pi 가 i 를 가리키므로 i 의 값을 3 으로 변경할 수 있겠네요. 따라서,

```
printf("호출 이후 i 의 값 : %d \n", i);
```

에는 i 의 값이 성공적으로 변경되어 3 이 출력되는 것입니다. 위 과정을 그림으로 나타내면 아래와 같습니다.



두 변수의 값을 교환하는 함수

```
/* 두 변수의 값을 교환하는 함수 */
#include <stdio.h>
int swap(int a, int b) {
  int temp = a;
```

```
  a = b;
  b = temp;

  return 0;
}
int main() {
  int i, j;

  i = 3;
  j = 5;

  printf("SWAP 이전 : i : %d, j : %d \n", i, j);

  swap(i, j);  // swap 함수 호출~~

  printf("SWAP 이후 : i : %d, j : %d \n", i, j);

  return 0;
}
```

성공적으로 컴파일 했으면

실행 결과

```
SWAP 이전 : i : 3, j : 5
SWAP 이후 : i : 3, j : 5
```

흠. 일단 우리가 원하던 결과가 나오지 않았습니다. 소스 상단의 주석에서도 볼 수 있듯이 swap 함수는 두 변수의 값을 교환해 주는 함수 입니다. 우리가 원하던 것은 SWAP 이후에 i 에는 5 가 j 에는 3 이 들어 있는 것인데 전혀 바뀌지 않았습니다. 이에 대해 이야기 하기 전에 소스 코드에서 보이는 새로운 것들에 대해 이야기 해봅시다.

```
int swap(int a, int b)
```

swap 함수의 정의를 보면 직관적으로 인자가 2 개나 있다는 것을 알 수 있습니다. 맞습니다. 이 swap 함수는 호출시 2 개의 인자를 전달해주어야 합니다. 물론 인자가 더 늘어난다면 반점(,) 을 이용해서 계속 늘려나가면 됩니다. 예를 들어서

```
int this_function_has_many_argumenets(int a, char b, int* k, long d, double c,
                                       int aa, char bb, int* kk, double cc)
```

와 같이요. 아무튼, swap 함수를 살펴 보면

```
int swap(int a, int b) {
  int temp = a;

  a = b;
  b = temp;

  return 0;
}
```

로 두 개의 int 형 인자를 받아 들이고 있습니다. 이 때, 내부를 보면 temp 라는 변수에 a 의 값을 저장합니다. 그리고 변수 a 에 변수 b 의 값을 넣습니다. 이제, 변수 b 에 a 의 값을 넣어야 하는데, 현재 변수 a 에는 b 의 값이 이미 들어가 있으므로 이전에 저장하였던 a 의 값인 temp 변수의 값을 b 에 넣으면 됩니다. 일단, 내용상으로는 전혀 하자가 없어 보입니다.

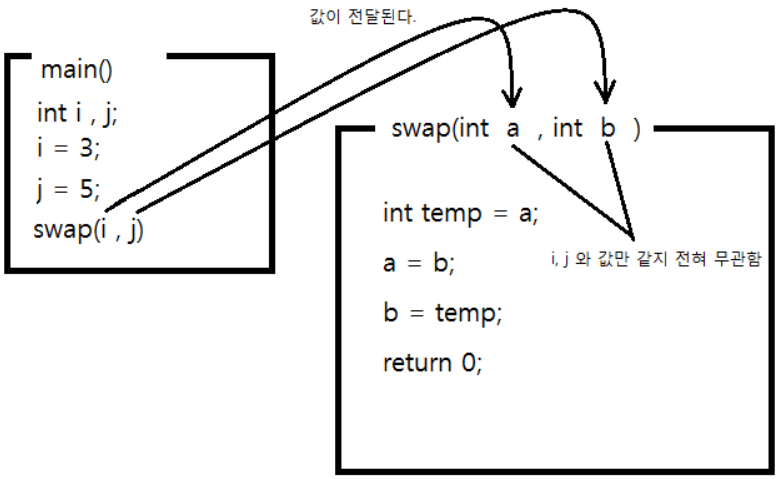
```
printf("SWAP 이전 : i : %d, j : %d \n", i, j);

swap(i, j);  // swap 함수 호출~~

printf("SWAP 이후 : i : %d, j : %d \n", i, j);
```

그런데, 말이죠. main 함수에서 i,j 의 값을 바꾸려고 swap 함수를 호출하였더니 전혀 뒤바뀌지 않은 채로 출력되었습니다. 도대체 왜 그런가요? 물론, 여러분은 다 알고 있겠지요. swap 함수의 변수 a,b 가 모두 swap 함수 내부에서 선언된 변수들이란 것입니다. 다시말해 변수 a 와 b 는 i 와 j 와 어떠한 연관도 없습니다. 다만, a 와 b 의 초기값이 i, j 와 동일하였다는 것만 빼고요.

이는 마치 아래의 작업을 한 것과 같습니다.

```
int i, j;
int temp, a, b;
/* 함수를 호출하여 함수의 인자를 전달하는 부분 */
a = i;
b = j;
/* 함수 몸체의 내용을 실행 */
temp = a;
a = b;
b = temp;
```

그러니 i 나 j 의 값이 바뀔리 만무하죠. 아무튼, 위 함수가 호출되는 과정을 그림으로 표현하면 아래와 같습니다.



그렇다면 어떻게 해야 할까요? 다 알고 있겠죠? 포인터를 이용합시다!

```c
/* 올바른 swap 함수 */
#include <stdio.h>
int swap(int *a, int *b) {
  int temp = *a;

  *a = *b;
  *b = temp;

  return 0;
}
int main() {
  int i, j;

  i = 3;
  j = 5;

  printf("SWAP 이전 : i : %d, j : %d \n", i, j);

  swap(&i, &j);

  printf("SWAP 이후 : i : %d, j : %d \n", i, j);

  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
SWAP 이전 : i : 3, j : 5
```

```
SWAP 이후 : i : 5, j : 3
```

오오오.. 드디어 우리가 원하던 것이 이루어졌습니다. 바로 i 와 j 의 값이 서로 뒤바뀐(swap) 것이지요. 아.. 정말 기쁩니다. 그런데, 이전에 이야기 하였던 내용을 잘 숙지하였더라면 위 함수가 왜 제대로 호출하는지 쉽게 알 수 있습니다.

```
int swap(int *a, int *b) {
  int temp = *a;

  *a = *b;
  *b = temp;

  return 0;
}
```

먼저 swap 함수를 살펴 봅시다. 이는 int 형을 가리키는 포인터 변수를 인자로 가지고 있습니다. 일단, swap 함수 내에서 두 변수를 교환하는 과정은 위와 동일하니 이에 대해서는 이야기 하지 않도록 하겠습니다. 이 때, main 함수에서는 swap 함수를 아래와 같이 호출합니다.

```
printf("SWAP 이전 : i : %d, j : %d \n", i, j);

swap(&i, &j);  // 호출

printf("SWAP 이후 : i : %d, j : %d \n", i, j);
```

바로 a 와 b 에 i 와 j 의 주소값을 전달하여 a 와 b 로 하여금 i 와 j 를 가리키게 만든 것입니다. 따라서, swap 함수 내부에서는 a 와 b 의 값을 교환하는 것이 아니라 a 와 b 가 가리키는 두 변수의 값을 교환했으므로 (*a, *b) 결과적으로 i 와 j 의 값이 바뀌게 된 것입니다. 어때요, 간단하지요?

결론적으로 정리하자면

> 어떠한 함수가 특정한 타입의 변수/배열의 값을 바꾸려면 함수의 인자는 반드시 그 타입을 가리키는 포인터를 이용해야 한다!

포인터를 인자로 받은 함수에 대해선 지속적으로 이야기 할 것이므로 지금 막상 이해가 잘 안된다고 해도 큰 걱정할 필요는 없습니다.

함수의 원형

우리가 여태까지 사용하였던 함수들은 모두 main 함수 위에서 정의되고 있었습니다. 그러면, 그 정의를 main 함수 아래에서 한다면 어떻게 될까요? 사실, 대부분의 사람들의 경우 main 함수를

제일 위에 놓고 나머지 함수들은 main 함수 뒤에 정의하게 됩니다. 아무튼, 위의 코드를 살짝 바꿔보면 아래와 같습니다.

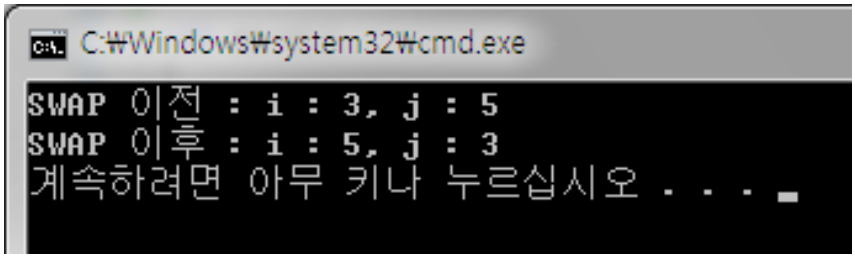
```c
/* 될까? */
#include <stdio.h>
int main() {
  int i, j;
  i = 3;
  j = 5;
  printf("SWAP 이전 : i : %d, j : %d \n", i, j);
  swap(&i, &j);
  printf("SWAP 이후 : i : %d, j : %d \n", i, j);

  return 0;
}
int swap(int *a, int *b) {
  int temp = *a;

  *a = *b;
  *b = temp;

  return 0;
}
```

컴파일 하게 되면 아래와 같은 경고 창을 볼 수 있습니다.

컴파일 오류

```
warning C4013: 'swap'이(가) 정의되지 않았습니다. extern은 int형을 반환하는
↪ 것으로 간주합니다.
```

흠, 일단은 무시하고 실행해 보도록 하죠.



어랏, 잘만 됩니다. 그렇다면 아무런 문제가 없는 것일까요? 사실, 함수를 적절히 잘 이용하기만 하면 큰 문제는 발생하지 않습니다. 그런데 말이죠. 사람도 역시 사람인지라, 프로그래밍 하다가 실수로 인자의 개수를 부족하게 쓰거나, 올바르지 않는 타입의 변수 (예를 들어 인자가 int* 인데, int 변수를 썼다든지) 를 사용하는 수가 발생하게 됩니다. 더군다나, 우리의 예제에서는 함수가

겨우 한 개 밖에 없었지만 실제 프로그래밍 시에는 수십개의 함수를 이용하기 때문이죠. 그렇다면, 여러분이 완벽한 인간이 아니라는 가정 하에 인자 하나를 누락시켜 봅시다.

위 코드의 함수 호출 부분을

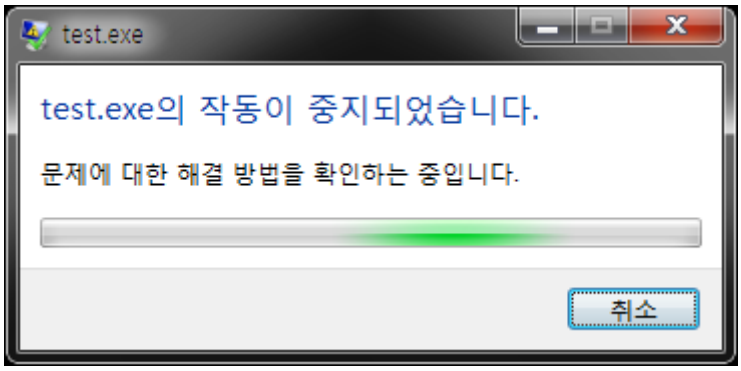
```
swap(&i, &j);
```

에서

```
swap(&i);
```

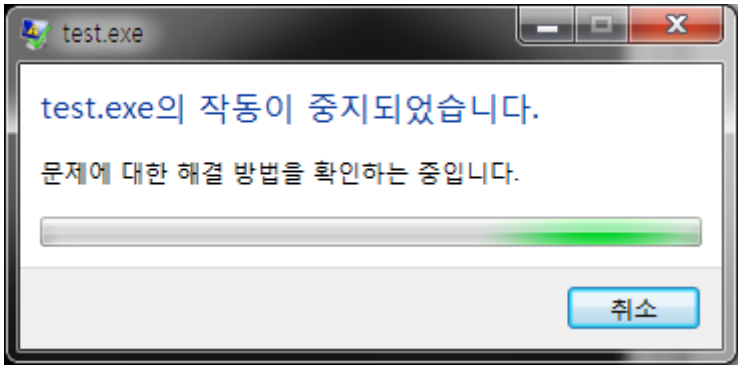
로 변경해봅시다.

컴파일 하면 여전히 위와 동일한 경고가 나오는데 특별히 내가 인자를 누락 했다는 말은 하지 않습니다. 그리고, 실행해보면



허걱! 컴파일시 아무런 오류 메세지도 없었는데 실행해 보면 위와 같이 덜컹 오류가 나타납니다. 이런 해괴한 일이 아닐 수 없군요. 게다가, 컴파일러는 내가 인자를 어디서 인자를 누락했는지 조차 표시해주지 않기 때문에 오류를 찾기 힘들어 질 수 밖에 없습니다. 물론, 우리의 예제는 짧기 때문에 찾기 쉽지만 진짜 같은 프로그램을 제작하면 코드가 보통 수천~수만 줄에 달한다는 것만을 기억하세요.

이번에는 swap 함수 호출 부분을 `swap(&i, j);` 로 변경해보세요.



뜨아아. 역시 이번에도 동일한 형태의 프로그램 오류가 컴파일시 오류 하나 없었는데도 불구하고 나타났습니다. 이 역시 포인터 전달 해야 되는데, 그냥 정수값을 전달해서 포인터 b 가 메모리의 올바르지 않은 공간에 접근하여 발생한 일입니다. 참으로 곤욕스러운 일이 아닐 수 없습니다. 우리가 아무리 대단하다고 해도 실수를 할 수 있는 법인데, 컴파일러는 이러한 실수를 하나도 잡아내지 못하고 있습니다.

그러나, 우리의 C 언어가 이를 용납할 수 있나요? C 언어에서는 멋진 해결책이 있습니다. 바로, 함수의 **원형(prototype)** 를 이용하는 것입니다.

```c
/* 함수의 원형 */
#include <stdio.h>
int swap(int *a, int *b);  // 이 것이 바로 함수의 원형
int main() {
  int i, j;
  i = 3;
  j = 5;
  printf("SWAP 이전 : i : %d, j : %d \n", i, j);
  swap(&i, &j);
  printf("SWAP 이후 : i : %d, j : %d \n", i, j);

  return 0;
}
int swap(int *a, int *b) {
  int temp = *a;

  *a = *b;
  *b = temp;

  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
SWAP 이전 : i : 3, j : 5
SWAP 이후 : i : 5, j : 3
```

오, 역시 잘 출력됩니다. 이번에는 컴파일시 경고나 오류의 흔적 조차 찾아볼 수 없었습니다.

```c
#include <stdio.h>
int swap(int *a, int *b);  // 이것이 바로 함수의 원형
int main() {
  int i, j;
  i = 3;
```

```
  j = 5;
  // ... (생략)
```

소스 코드의 제일 윗부분을 보면 위와 같이 한 줄이 추가된 것을 볼 수 있습니다. 이는 바로 '함수의 원형' 이라 부르는 것입니다. 이는 사실 함수의 정의 부분을 한 번 더 써준 것 뿐입니다 (주의할 점은 함수의 원형에는 정의와는 달리 뒤에 ; 를 붙인 다는 것입니다). 그런데, 이 한줄이 컴파일러에게 다음과 같은 사실을 알려줍니다.

> 야, 이 소스코드에 이러 이러한 함수가 정의되어 있으니까 잘 살펴봐

다시말해, 컴파일러에게 이 소스코드에 사용되는 함수에 대한 정보를 제공하는 것입니다. 다시 말해 실제 프로그램에는 전혀 반영되지 않는 정보지요. 그렇지만, 우리가 앞서 하였던 실수들을 하지 않도록 도와줍니다. 만일, 위와 같이 함수의 원형을 삽입한 상태에서 인자를 `&i` 하나로 지워 봅시다. 즉, `swap(&i, &j)` 를 `swap(&i);` 로 변경해봅시다.

그럼 컴파일 시 아래와 같은 오류를 만나게 됩니다.

컴파일 오류

```
error C2198: 'swap' : 호출에 매개 변수가 너무 적습니다.
```

와우! 우리가 앞서 함수의 원형을 집어 넣지 않았을 때 에는 인자(매개 변수)를 하나 줄여도 아무말 하지 않던 컴파일러가 원형을 삽입하고 나니 위와 같이 정확한 위치에 내가 어딜 잘못했는지 잡아냅니다. 이것이 가능한 이유가 바로 컴파일러에게 내가 무슨 무슨 함수를 이용할 것인지 함수의 원형을 통해 이야기 하였기 때문입니다. 내가, `int swap(int *a, int *b)` 라는 함수가 있다는 것을 원형을 이용해 알려주었기 때문에 컴파일러는 우리가 `swap` 함수를 사용하면 꼭 2 개의 인자를 이용한다는 사실을 알게 되어 내가 인자를 하나만 적었을 때 틀렸다고 알려 준 것입니다.

그렇다면 `swap(&i, &j)` 를 `swap(&i, j)` 로 바꿔보면 어떻게 될까요?

컴파일 오류

```
warning C4047: '함수' : 'int *'의 간접 참조 수준이 'int'과(와) 다릅니다.
warning C4024: 'swap' : 형식 및 실제 매개 변수 2의 형식이 서로 다릅니다.
```

실질적인 오류는 발생하지 않았지만 일단, 내가 잘못하였다는 것을 알려줍니다. 컴파일러는 역시 원형을 통해 두 번째 매개 변수의 타입이 무엇인지 알고 있기에 그냥 `int` 를 사용하면 함수의 두번째 매개변수와 내가 인자에 전달하는 변수의 형과 다르다는 사실을 알려 줍니다. 다만, 여기서 아까와 같이 오류가 출력되지 않는 이유는 `int*` 도 사실 `int` 형 데이터 이기 때문에 `j` 가 `(int *)` 로 캐스팅 되어 전달되므로, 아까와 같은 강한 오류 메세지는 출력되지 않습니다. 그러나, 여전히 프로그래머의 잘못을 지적하고 있습니다.

이러한 연유에서, 함수의 원형을 집어넣는 일은 여러분들이 '반드시' 하셔야 되는 일입니다. 물론, `main` 함수 위에 함수를 정의하면 상관 없지만 사실 `99.9%` 의 프로그래머들은 함수를 `main` 함수의 뒤에 정의하고 원형을 앞에 추가하는 것을 선호하니 여러분들도 트렌드를 따르시기 바랍니다.

배열을 인자로 받기

이번에는 배열을 인자로 받아 들이는 함수에 대해서 생각해봅시다. 이번 예제에서 우리가 만들게 된 함수는 바로, 배열을 인자로 받아서 그 배열의 각 원소의 값을 1 씩 증가시키는 함수 입니다.

```c
#include <stdio.h>

int add_number(int *parr);
int main() {
  int arr[3];
  int i;

  /* 사용자로 부터 3 개의 원소를 입력 받는다. */
  for (i = 0; i < 3; i++) {
    scanf("%d", &arr[i]);
  }

  add_number(arr);

  printf("배열의 각 원소 : %d, %d, %d", arr[0], arr[1], arr[2]);

  return 0;
}
int add_number(int *parr) {
  int i;
  for (i = 0; i < 3; i++) {
    parr[i]++;
  }
  return 0;
}
```

성공적으로 컴파일 했으면

실행 결과

```
10
11
15
배열의 각 원소 : 11, 12, 16
```

음, 역시 함수가 잘 작동하는 군요. 우리가 `10, 11, 15` 를 입력했을 때, 함수를 통해서 각 원소가 1 씩 증가하여 `11, 12, 16` 이 되었습니다. 일단, `add_number` 함수 부터 살펴 보도록 하죠.

```
int add_number(int *parr)
```

우리가, 앞서 말한 내용에 따르면 '특정한 타입의 값을 변경하는 함수를 제작하려면, 반드시 그 타입을 가리키는 포인터를 인자로 가져야 한다' 라고 했습니다. 그렇다면, 우리가 `arr` 이라는 배열을 가리키는 포인터가 바로 `add_number` 의 인자로 와야 하는데, 우리가 `12 - 3` 강에서 배운 내용에 따르면 `int arr[3]` 와 같은 일차원 배열을 가리키는 포인터는 바로 `int*` 형이라 했습니다. (잘 모르겠다면 여기를 눌러서 강의를 다시 보시기 바랍니다)

따라서, `add_number(int *parr)` 이라 하면 `arr` 을 가리키도록 인자를 받을 수 있습니다. 함수를 호출 할 때 아래와 같이 하였습니다.

```
add_number(arr);
```

그런데, 우리가 이전에 배운 바에 따르면 `arr` 은 배열의 시작 주소 값을 가지고 있다고 하였습니다. 즉, `arr = &arr[0]` 인 것이지요. 따라서, `parr` 에는 `arr` 배열의 시작 주소, 즉 배열 `arr` 을 가리키게 됩니다.

```
{
  int i;
  for (i = 0; i < 3; i++) {
    parr[i]++;
  }
  return 0;
}
```

마지막으로 함수의 몸체를 살펴봅시다. `parr[i]` 를 통해 `parr` 이 가리키는 배열의 `(i + 1)` 번째 원소에 접근할 수 있습니다 (`arr[1]` 이 배열의 두 번째 원소 이므로). 따라서, `parr[i]++` 을 통해서 배열의 각 원소들의 크기를 모두 1 씩 증가시키게 됩니다. 사실, 위 함수가 어떻게 돌아가는지 잘 이해하기 위해서는 포인터와 배열에 대한 거의 완벽한 이해를 필요로 합니다. 만약 그러지 않는다면 모래사장에 빌딩 짓는 것처럼, C 언어에 대한 개념을 완전히 잊어버릴 수 있으니 모른 다면 꼭 뒤로 가기를 하여 복습을 하시기 바랍니다.

```
/* 입력 받은 배열의 10 개의 원소들 중 최대값을 출력 */
#include <stdio.h>
/* max_number : 인자로 전달받은 크기 10 인 배열로 부터 최대값을 구하는 함수 */
int max_number(int *parr);
int main() {
  int arr[10];
  int i;
```

```
  /* 사용자로 부터 원소를 입력 받는다. */
  for (i = 0; i < 10; i++) {
    scanf("%d", &arr[i]);
  }

  printf("입력한 배열 중 가장 큰 수 : %d \n", max_number(arr));
  return 0;
}
int max_number(int *parr) {
  int i;
  int max = parr[0];

  for (i = 1; i < 10; i++) {
    if (parr[i] > max) {
      max = parr[i];
    }
  }

  return max;
}
```

성공적으로 컴파일 한다면

실행 결과

```
100 50 102 300 900 700 550 400 800 600
입력한 배열 중 가장 큰 수 : 900
```

이번 예제는 사용자들로 부터 정수 10 개를 입력 받아서, 그 수들 중 가장 큰 수를 뽑아내는 프로그램입니다. 먼저 max_number 함수 부터 살펴봅시다.

```
int max_number(int *parr) {
  int i;
  int max = parr[0];

  for (i = 1; i < 10; i++) {
    if (parr[i] > max) {
      max = parr[i];
    }
  }

  return max;
}
```

음.. 일단, 소스 코드를 해석하는데에는 큰 어려움이 없을 것 같네요. 일단 처음 max 에 parr 이 가리키는 배열의 [0], 즉 첫번째 원소의 값을 넣었습니다. 그리고 아래 for 문에서 만약 parr[i]

가 max 보다 크면 max 의 값을 parr[i] 로 대체하고 있군요. 결과적으로 i 값이 9 까지 되었을 때에는 max 에 parr 중 가장 큰 원소의 값이 들어가게 됩니다. 만일, max 보다 더 큰 원소가 parr 에 있다면 max 의 값은 그 큰 원소의 값으로 바뀌었기 때문에 모순이지요.

결과적으로 우리가 입력한 10 개의 원소들 중 가장 큰 원소가 출력됩니다.

함수 사용 연습하기

사실, 아직까지 함수가 왜 이리 중요한 것인지 감이 잘 오지 않는 분들이 있을 것입니다. 그래서, 그러하신 분들을 위해 함수의 중요성을 절실히 느낄 수 있는 예제를 준비하였습니다.

다음의 두 소스 코드를 비교해 보면서 어떤 것이 나은지 생각해보세요

```c
/* 함수를 이용하지 않은 버전 */
#include <stdio.h>
int main() {
  char input;

  scanf("%c", &input);

  if (48 <= input && input <= 57) {
    printf("%c 는 숫자 입니다 \n", input);
  } else {
    printf("%c 는 숫자가 아닙니다 \n", input);
  }

  return 0;
}
```

```c
/* 함수를 이용한 버전 */
#include <stdio.h>
int isdigit(char c);  // c 가 숫자인지 아닌지 판별하는 함수
int main() {
  char input;

  scanf("%c", &input);

  if (isdigit(input)) {
    printf("%c 는 숫자 입니다 \n", input);
  } else {
    printf("%c 는 숫자가 아닙니다 \n", input);
  }

  return 0;
}
int isdigit(char c) {
  if (48 <= c && c <= 57) {
```

```
    return 1;
  } else
    return 0;
}
```

일단, 첫번째 소스의 경우 길이가 짧습니다. 다만 이해하기가 힘듦니다.

```
if (48 <= input && input <= 57) {
}
```

`printf` 문이 없다고 했을 때 위 코드가 `input` 이 숫자인지 아닌지 판별하는지 쉽게 구분이 가나요? 이는 특별히 주석을 넣지 않는 한 매우 어렵습니다. 사실, 숫자의 경우 아스키 코드의 값이 48 에서 57 이기 때문에 위 코드를 사용하였는데 아스키 코드표를 외우고 다니지 않는 한 이해하기 상당히 어렵습니다.

그렇다면 함수를 이용한 버전을 살펴 봅시다.

```
if (isdigit(input)) {
}
```

일단 `isdigit` 라는 이름만 보고도 이 함수는 `input` 이 숫자 인지 아닌지 (is digit?) 판별하는 함수 임을 알 수 있습니다. 물론, `isdigit` 함수 내부에도 첫번째 소스와 동일한 과정이 진행되지만 이 함수가 무슨 작업을 하는지 알기 때문에 소스를 이해하기 훨씬 쉬워집니다. 뿐만 아니라, 어떠한 문자가 숫자인지 반복해서 확인하는 경우에도 함수를 이용하면 편히 사용할 수 있습니다.

아직까지도 왜 함수를 써야 하는지 모르겠다고 해서 큰 문제는 아닙니다. 나중에 가면 자연스럽게 깨닫게 될 것입니다. 그럼, 이번 강의는 여기까지에서 줄이겠습니다.

생각 해보기

문제 1

위 10 개의 원소들 중 최대값 구하는 함수를 이용하여, 10 개의 원소를 입력 받고 그 원소를 큰 순으로 출력하는 함수를 만들어보세요. (난이도 : 中)

문제 1

2 차원 배열의 각 원소에 1 을 더하는 함수의 인자는 어떤 모양일까요? (난이도 : 中下 2 차원 배열의 각 원소에 1 을 더하는 함수의 인자는 어떤 모양일까요? (난이도 : 中下)

여러가지 인자들

와.. 드디어, 함수만 세번째 강의입니다. 아마 이전 강좌에서 배운 내용들 중 어려운 것은 없으리라 생각됩니다. 물론, 이번 강좌의 내용도 이전까지의 내용을 잘 숙지 하셨더라면 무난하게 넘어갈 수 있으리라 생각됩니다.

지난번 내용을 상기해보며

지난번 내용은 잘 기억하고 있는지요? 다시 한 번 요약해 보자면, "어떠한 함수가 특정한 타입의 변수/배열의 값을 바꾸려면 함수의 인자는 반드시 타입을 가리키는 포인터 형을 이용해야 한다!" 였습니다.사실, 이 문장이 이해가 잘 되지 않았던 분들이 있으리라 생각됩니다. 하지만, 이번 강좌를 보고 난다면 이 문장의 의미를 정확하게 파악할 수 있을 것입니다.

```c
/* 눈 돌아가는 예제. 포인터가 가리키는 변수를 서로 바꾼다.  */
#include <stdio.h>

int pswap(int **pa, int **pb);
int main() {
  int a, b;
  int *pa, *pb;

  pa = &a;
  pb = &b;

  printf("pa 가 가리키는 변수의 주소값 : %p \n", pa);
  printf("pa 의 주소값 : %p \n \n", &pa);
  printf("pb 가 가리키는 변수의 주소값 : %p \n", pb);
  printf("pb 의 주소값 : %p \n", &pb);

  printf(" -------------- 호출 -------------- \n");
  pswap(&pa, &pb);
  printf(" -------------- 호출끝 -------------- \n");

  printf("pa 가 가리키는 변수의 주소값 : %p \n", pa);
  printf("pa 의 주소값 : %p \n \n", &pa);
  printf("pb 가 가리키는 변수의 주소값 : %p \n", pb);
  printf("pb 의 주소값 : %p \n", &pb);
  return 0;
}
int pswap(int **ppa, int **ppb) {
  int *temp = *ppa;

  printf("ppa 가 가리키는 변수의 주소값 : %p \n", ppa);
  printf("ppb 가 가리키는 변수의 주소값 : %p \n", ppb);
```

```
  *ppa = *ppb;
  *ppb = temp;

  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
pa 가 가리키는 변수의 주소값 : 0x7ffc5ffd7520
pa 의 주소값 : 0x7ffc5ffd7528

pb 가 가리키는 변수의 주소값 : 0x7ffc5ffd7524
pb 의 주소값 : 0x7ffc5ffd7530
 ------------- 호출 --------------
ppa 가 가리키는 변수의 주소값 : 0x7ffc5ffd7528
ppb 가 가리키는 변수의 주소값 : 0x7ffc5ffd7530
 ------------- 호출끝 --------------
pa 가 가리키는 변수의 주소값 : 0x7ffc5ffd7524
pa 의 주소값 : 0x7ffc5ffd7528

pb 가 가리키는 변수의 주소값 : 0x7ffc5ffd7520
pb 의 주소값 : 0x7ffc5ffd7530
```

여러분의 출력결과는 위 사진과 다를 수 있습니다.

pa 가 가리키는 변수의 주소값은 (즉, pa 의 값이지요) 0x7520 였습니다. (편의상 앞에 공통된 0x7ffc5ffd 는 생략합시다) pb 가 가리키는 변수의 주소값은 0x7524 이였습니다. 그런데 말이죠. pswap 함수를 호출하고 나니, pa 가 가리키는 변수의 주소값은 0x7524 이 되고, pb 가 가리키는 변수의 주소값은 0x7520 가 되었습니다. 즉, 두 포인터가 가리키는 변수가 서로 뒤바뀐 것이지요.

이 때, 우리는 이와 같은 함수를 만들기 위해서, 인자를 어떤 형식으로 취해야 될까요? 앞서 말했듯이, 특정한 타입의 변수의 값을 바꾸려면, 특정한 타입을 가리키는 포인터로 인자를 취해야 된다고 했습니다. 그런데, 이 예제의 경우, 특정한 타입은 int* 타입입니다. 그렇다면 int* 타입을 가리키는 포인터의 타입은? 음. 강좌를 잘 복습하였다면 int** 타입 이라고 말할 수 있겠지요. (잘 모르겠다면 12-3 강, 포인터는 영희이다!를 보세요)

따라서, 우리는 위 이야기를 토대로 아래와 같이 함수를 정의하였습니다.

```
int pswap(int **ppa, int **ppb)
```

상당히, 잘한 것이지요. 이제, 함수의 몸체를 봐봅시다.

```
int pswap(int **ppa, int **ppb) {
  int *temp = *ppa;

  printf("ppa 가 가리키는 변수의 주소값 : %p \n", ppa);
  printf("ppb 가 가리키는 변수의 주소값 : %p \n", ppb);

  *ppa = *ppb;
  *ppb = temp;

  return 0;
}
```

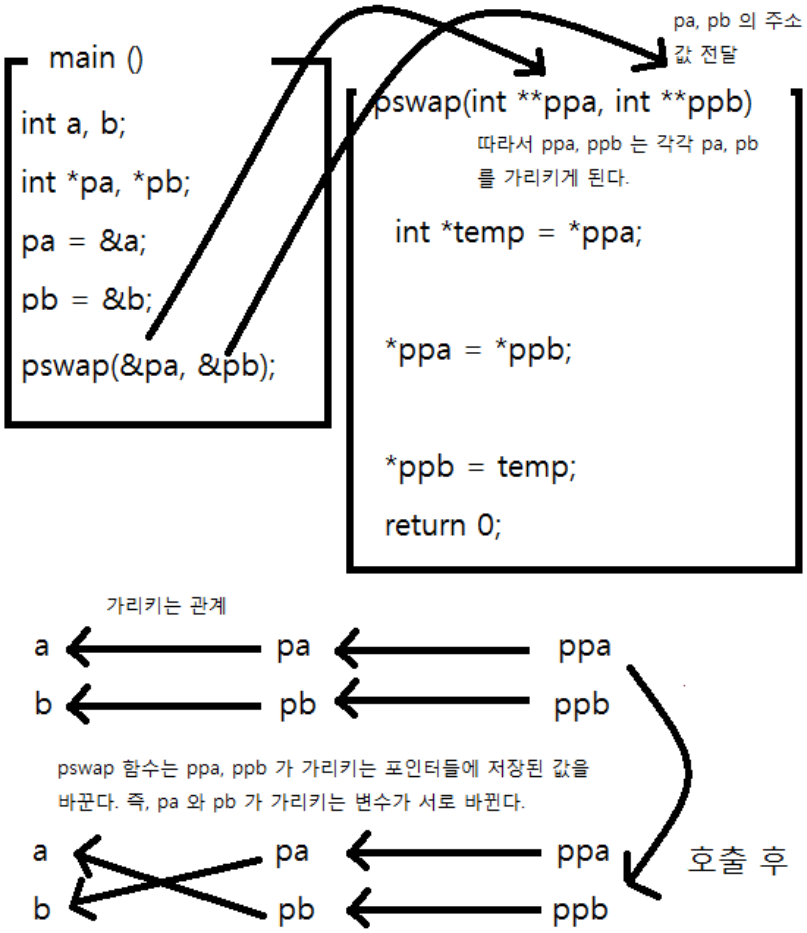
일단, int* 형의 temp 변수를 만들어서 *ppa 의 값을 저장하고 있습니다. 그런데, *ppa 의 값은 무엇일까요?

만일 우리가 위 예제 처럼 pswap 함수를 호출하였다고 하면, ppa 는 pa 를 가리키고 있고, ppb 는 pb 를 가리키고 있겠지요. 따라서, *ppa 라 하면 pa 의 값을 이야기 합니다. 그런데, pa 는 int* 형이므로, pa 의 값을 보관하는 변수는 반드시 int* 이여야 되겠지요. 따라서, 우리는 int* 형의 temp 변수를 정의하였습니다. 그 아래의 내용은 이전에 만들어 보았던 swap 함수와 동일합니다.

```
printf("ppa 가 가리키는 변수의 주소값 : %p \n", ppa);
printf("ppb 가 가리키는 변수의 주소값 : %p \n", ppb);
```

그렇다면 우리는 위 두개의 printf 문장에서 어떤 결과가 출력될 지 예측 가능합니다. 위 예제에서 ppa 가 pa 를 가리키고 있으므로 ppa 의 값을 출력하면 pa 의 주소값이 나오고, ppb 도 마찬가지로 나오겠죠. 위 출력결과에서 실제로 같다는 것을 확인할 수 있습니다. 어때요. pswap 함수가 이해가 되나요?

위 과정을 그림으로 표현하면 아래와 같습니다.



그렇다면, 이번에는 이차원 배열을 인자로 받는 함수에 대해서 생각해 보도록 합시다.

```c
/* 2 차원 배열의 각 원소를 1 씩 증가시키는 함수 */
#include <stdio.h>
/* 열의 개수가 2 개인 이차원 배열과, 총 행의 수를 인자로 받는다. */
int add1_element(int (*arr)[2], int row);
int main() {
  int arr[3][2];
  int i, j;

  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      scanf("%d", &arr[i][j]);
    }
  }

  add1_element(arr, 3);

  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      printf("arr[%d][%d] : %d \n", i, j, arr[i][j]);
```

```
    }
  }
  return 0;
}
int add1_element(int (*arr)[2], int row) {
  int i, j;
  for (i = 0; i < row; i++) {
    for (j = 0; j < 2; j++) {
      arr[i][j]++;
    }
  }

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
1 2 3 4 5 6
arr[0][0] : 2
arr[0][1] : 3
arr[1][0] : 4
arr[1][1] : 5
arr[2][0] : 6
arr[2][1] : 7
```

역시 잘 실행되는군요. 일단, 함수의 정의부분 부터 살펴봅시다.

```
int add1_element(int (*arr)[2], int row) {
  int i, j;
  for (i = 0; i < row; i++) {
    for (j = 0; j < 2; j++) {
      arr[i][j]++;
    }
  }

  return 0;
}
```

이 함수는 인자를 두 개 받고 있는데 하나는 열의 개수가 2 개인 이차원 배열을 가리키는 포인터이고, 하나는 함수의 행의 수를 받는 인자입니다.

```
for (i = 0; i < row; i++) {
  for (j = 0; j < 2; j++) {
```

```
    arr[i][j]++;
  }
}
```

우리는 row 를 통해 이 이차원배열의 행의 개수를 알 수 있고, 열의 개수는 이미 알고 있으므로 (배열 포인터에서) 각 원소를 1 씩 증가시키는 작업을 시행할 수 있게됩니다. 위와 같이 말이죠. 우리는 포인터를 잘 배워서 헷갈릴 문제는 없지만 많은 사람들에게 다음과 같이 인자를 받는것이 어렵게 느껴집니다.

```
int add1_element(int (*arr)[2], int row)
```

그래서, 오직 함수의 인자의 경우에서만 위 형태의 인자를 다음과 같이도 표현할 수 있습니다.

```
int add1_element(int arr[][2], int row)
```

이는 오직 함수의 인자에서만 적용되는 것입니다. 만일

```
int parr[][3] = arr;
```

와 같은 문장을 이용했더라면 컴퓨터는 parr 을 '열의 개수가 3 개이고 행의 개수는 정해지지 않는 배열' 이라 생각해서 오류를 내게 됩니다. (만일 행의 개수를 생략했다면 배열을 정의시 초기화도 해주어야 되는데는 위는 그러지 않으므로) 암튼, 함수의 인자에서만 가능한 형태라는 것을 기억해 주시기 바랍니다.

덧붙여서 응용력을 살짝 이용하면 다차원 배열의 인자도 정의할 수 있습니다. 예를 들어서

```
int multi(int (*arr)[3][2][5]) {
  arr[1][1][1][1] = 1;
  return 0;
}
```

혹은

```
int multi(int arr[][3][2][5]) {
  arr[1][1][1][1] = 1;
  return 0;
}
```

로 하면 됩니다.

상수인 인자

```c
/* 상수를 인자로 받아들이기 */
#include <stdio.h>
int read_val(const int val);
int main() {
  int a;
  scanf("%d", &a);
  read_val(a);
  return 0;
}
int read_val(const int val) {
  val = 5;  // 허용되지 않는다.
  return 0;
}
```

컴파일 하게 되면 아래와 같은 오류를 만나게 됩니다.

컴파일 오류

```
error C2166: l-value가 const 개체를 지정합니다.
```

흠.. 이건 우리가 이전에 상수의 값을 변경하려고 했었을 때 만났던 오류 인것 같습니다. 맞습니다. 우리가 `val` 을 `const int` 로 선언하였기 때문에 함수를 호출 할 때, `val` 의 값은 인자로 전달된 값으로 초기화 되고 결코 바뀌지 않습니다. 즉, `val` 은 `a` 의 값으로 상수로 초기화 된 것입니다. 따라서, 함수 내부에서 `val = 5` 와 같이 `val` 의 값을 바꾸려 한다면 오류가 나겠지요. 왜냐하면 `val` 은 상수이니까요.

상수로 인자를 받아들이는 경우 대부분은 함수를 호출 해도 그 인자의 값이 바뀌지 않는 경우에 자주 사용합니다만, 자세한 내용은 나중에 좀더 다루도록 하겠습니다.

함수 포인터

아마, '함수 포인터' 라는 말을 들었을 때는 조금 의아하는 감이 있지 않을까 합니다. 함수 포인터라니, 함수를 가리킨다는 것인가? 그럼, 함수가 메모리 상에 있다는 거야? 네. 맞습니다. 사실, 프로그램의 코드 자체가 메모리 상에 존재합니다. 우리는 이전에 컴파일러가 하는 작업이 바로 우리가 '인간에 친숙한 언어' 로 쓰여진 프로그램 코드를 '컴퓨터에 친숙한 언어, 즉 수 데이터들' 로 바꿔주어 실행 파일을 생성한다고 배웠습니다. 이렇게, 바뀐 실행 파일을 실행하게 되면 프로그램의 수 코드가 메모리 상에 올라가게 됩니다. 다시말해, 메모리 상에 함수의 코드가 들어간다는 것입니다. 이 때, 변수를 가리키는 포인터 처럼 함수 포인터는 메모리 상에 올라간 함수의 시작 주소를 가리키는 역할을 하게 됩니다.

그렇다면, 함수 포인터가 함수를 가리키기 위해서는 그 함수의 시작 주소값을 알아야 합니다. 그런데, 배열과 마찬가지로 함수의 이름이 바로 함수의 시작 주소값을 나타냅니다.

```c
/* 함수 포인터 */
#include <stdio.h>

int max(int a, int b);
int main() {
  int a, b;
  int (*pmax)(int, int);
  pmax = max;

  scanf("%d %d", &a, &b);
  printf("max(a,b) : %d \n", max(a, b));
  printf("pmax(a,b) : %d \n", pmax(a, b));

  return 0;
}
int max(int a, int b) {
  if (a > b)
    return a;
  else
    return b;

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
10 15
max(a,b) : 15
pmax(a,b) : 15
```

역시 우리가 예상했던 데로 잘 흘러가는 것 같습니다. 함수 포인터는 어떻게 정의하는지 살펴봅시다.

```c
int (*pmax)(int, int);
```

일단, 위는 함수 포인터 pmax 의 정의 입니다. 위 정의를 보고 다음과 같은 사실을 알 수 있습니다. '이 함수 포인터 pmax 는 함수의 리턴값이 int 형이고, 인자 두 개가 각각 int 인 함수를 가리키는구나!'. 따라서, 우리는 pmax 함수 포인터로 특정한 함수를 가리킬 때, 그 함수는 반드시 pmax 의 정의와 일치해야 합니다. 함수 포인터의 일반적인 정의는 다음과 같습니다.

(함수의 리턴형) (*포인터 이름)(첫번째 인자 타입, 두번째 인자 타입,....) // 만일 인자가 없다면 그냥 괄호 안을 비워두면 된다. 즉, int (*a)() 와 같이 하면 된다

이제 pmax 가 max 를 가리키게 되는 부분을 봅시다.

```
pmax = max;
```

max 함수를 살펴보면 pmax 의 정의와 일치하므로, max 함수의 시작 주소값을 pmax 에 대입할 수 있게 됩니다. 이 때, 앞에서도 말했듯이 특정한 함수의 시작 주소값을 알려면 그냥 함수 이름을 넣어주면 됩니다. pmax = &max 와 같은 형식은 틀린 것입니다.

```
printf("max(a,b) : %d \n", max(a, b));
printf("pmax(a,b) : %d \n", pmax(a, b));
```

pmax 는 이제 max 함수를 가리키므로 pmax 를 통해 max 함수가 할 수 있었던 모든 작업들을 할 수 있게 됩니다. 이때도 역시 그냥 pmax 를 max 처럼 이용하면 됩니다. 이는 배열에서

```
int arr[3];
int *p = arr;

arr[2];  // p[2] 와 정확히 일치
p[2];
```

와 같이 arr[2] 와 p[2] 가 동일한 것과 같습니다. 아무튼 max(a,b) 를 하나 pmax(a,b) 를 하나 결과는 똑같이 나오게 됩니다.

```
/* 함수 포인터 */
#include <stdio.h>

int max(int a, int b);
int donothing(int c, int k);
int main() {
  int a, b;
  int (*pfunc)(int, int);
  pfunc = max;

  scanf("%d %d", &a, &b);
  printf("max(a,b) : %d \n", max(a, b));
  printf("pfunc(a,b) : %d \n", pfunc(a, b));

  pfunc = donothing;
```

```
  printf("donothing(1,1) : %d \n", donothing(1, 1));
  printf("pfunc(1,1) : %d \n", pfunc(1, 1));
  return 0;
}
int max(int a, int b) {
  if (a > b)
    return a;
  else
    return b;

  return 0;
}
int donothing(int c, int k) { return 1; }
```

성공적으로 컴파일 했다면

실행 결과

```
10 123
max(a,b) : 123
pfunc(a,b) : 123
donothing(1,1) : 1
pfunc(1,1) : 1
```

일단, 우리는 이전의 예제와 동일한 형태의 함수 포인터 pfunc 을 정의하였습니다.

```
int (*pfunc)(int, int);
```

이는 '리턴형이 int 이고 두 개의 인자 각각의 포인터 형이 int 인 함수를 가리킵니다. 그런데, donothing 함수와 max 함수 모두 이 조건을 만족하고 있습니다. 즉, 이들은 인자의 변수들도 다루고 하는 일도 다르지만 리턴값이 int 로 같고 두 개의 인자 모두 int 이므로 pfunc 이 이 두개의 함수를 가리킬 수 있는 것입니다.

```
pfunc = max;

scanf("%d %d", &a, &b);
printf("max(a,b) : %d \n", max(a, b));
printf("pfunc(a,b) : %d \n", pfunc(a, b));

pfunc = donothing;

printf("donothing(1,1) : %d \n", donothing(1, 1));
printf("pfunc(1,1) : %d \n", pfunc(1, 1));
```

따라서, 위와 같이 했을 때 pfunc 이, 자기가 가리키는 함수의 역할을 제대로 하고 있다는 것을 알 수 있습니다. 그런데 말이죠. 함수 포인터를 만들 때, 인자의 형이 무엇인지 알기 힘든 경우가 종종 있습니다. 예를 들어 아래와 같은 함수의 원형을 봅시다.

```
int increase(int (*arr)[3], int row)
```

흠... 두 번째 인자의 형은 int 라는 것은 알겠는데 첫번째 인자의 형은 도대체 뭘까요? 사실, 간단합니다. 특정한 타입의 인자를 판별하는 일은 단순히변수의 이름만을 빼버리면 됩니다. 따라서, 첫번째 인자의 형은 int (*)[3] 입니다. 즉, increase 함수를 가리키는 함수 포인터의 원형은 아래와 같습니다.

```
int (*pfunc)(int (*)[3], int);
```

간단하지요? 이것을 이전에 이차원 배열을 인자로 받았던 함수에 적용시켜 보면 정확히 작동한다는 것을 알 수 있습니다.

그럼, 이번 강좌는 여기에서 끝을 내도록 하겠습니다. 함수에 관한 강좌는 여기서 막을 내리게 됩니다. 사실, 아직까지도 C 언어를 배우면서 정말로 무언가 할 수 있는 실용적인 프로그램을 만들지 못해서 안타깝습니다. 그래서 이번에 생각해보기로 여러 재미있는 과제들을 내보도록 하죠.

생각해보기

문제 1

사용자로 부터 5 명의 학생의 수학, 국어, 영어 점수를 입력 받아서 평균이 가장 높은 사람 부터 평균이 가장 낮은 사람까지 정렬되어 출력하도록 하세요. 특히, 평균을 기준으로 평균 이상인 사람 옆에는 '합격', 아닌 사람은 '불합격' 을 출력하게 해보세요 (난이도 : 中上).

문제 2

유클리도 호제법을 이용해서 N 개의 수들의 최대공약수를 구하는 함수를 만들어보세요. 유클리드 호제법이 무엇인지 모르신다면, 인터넷 검색을 활용하는 것을 추천합니다. (댓글을 달아도 돼요) (난이도 : 中上)

문제 3

자기 자신을 호출하는 함수를 이용해서 1 부터 특정한 수까지의 곱을 구하는 프로그램을 만들어보세요. (난이도 : 下)

문제 4

계산기를 만들어보세요. 사용자가 1 을 누르면 +, 2 를 누르면 - 와 같은 방식으로 해서 만들면 됩니다. 물론 이전의 계산 결과는 계속 누적되어야 하고, 지우기 기능도 있어야 합니다. (물론 하나의 함수에 구현하는 것이 아니라 여러개의 함수로 분할해서 만들어야겠죠?) (난이도 : 中)

문제 5

N 진법에서 M 진법으로 변환하는 프로그램을 만들어보세요. (난이도 : 中)

문제 6

에라토스테네스의 체를 이용해서 1 부터 N 까지의 소수를 구하는 프로그램을 만들어보세요. (난이도 : 中)

문제 7

1000 자리의 수들의 덧셈, 뺄셈, 곱셈, 나눗셈을 수행하는 프로그램을 만들어보세요. 나눗셈의 경우 소수 부분을 잘라버리세요. 물론, 소수 부도 1000 자리로 구현해도 됩니다. 1000 자리 수들의 연산 수행 시간은 1 초 미만이여야 합니다. (난이도 : 上)

생각해볼 문제에 대한 아이디어

안녕하세요 여러분. 이전 강좌의 예제가 중요한 만큼, 예제에 좀더 쉽게 접근할 수 있는 아이디어에 관해서 짧은 힌트 형식으로 강좌를 작성하도록 하겠습니다. 물론, 언제까지나 힌트일 뿐 완전한 코드는 여러분이 완성시켜야 합니다.

생각해볼 문제 1

사용자로 부터 5 명의 학생의 수학, 국어, 영어 점수를 입력 받아서평균이 가장 높은 사람 부터 평균이 가장 낮은 사람까지 정렬되어 출력하도록 하세요. 특히, 평균을 기준으로 평균 이상인 사람옆에는 '합격', 아닌 사람은 '불합격' 을 출력하게 해보세요.

일단, 여러분이 직면했을 가장 큰 문제는 '어떻게 정렬하는 프로그램' 을 만드느냐 이였겠습니다. 정렬을 하는 방법 (보통 알고리즘 이라 표현합니다) 에는 여러가지가 있습니다. 가장 직관적으로 이해하기 쉬운 것은버블 정렬(Bubble sorting) 이라 불리는 것인데 컴퓨터가 다음과 같은 규칙을 통해 작업을 합니다.

예를 들어 5, 1, 4, 2, 8 을 정렬한다고 합시다.

(**5** **1** 4 2 8) -> (**1** **5** 4 2 8)

버블 정렬 알고리즘은 처음 두 개의 원소를 비교해 왼쪽이 크면 자리를 바꿉니다. 이 경우, 5 가 1 보다 더 크기 때문에 1 과 5 의 자리를 바꾸었습니다.

(1 **5** **4** 2 8) -> (1 **4** **5** 2 8)

그 다음 두 원소를 비교합니다. 이번에도 5 가 더 크므로 4 와 자리를 바꿉니다.

(1 4 **5** **2** 8) -> (1 4 **2** **5** 8)

그 다음 두 원소 5,2 를 비교합니다. 이번에도 5 가 더 크므로 자리를 바꿉니다.

(1 4 2 **5** **8**) -> (1 4 2 **5** **8**)

그 다음 두 원소 5,8 를 비교합니다. 이번에는 오른쪽이 더 크므로 자리를 안바꿔도 됩니다. 끝 원소까지 비교하였다면, 가장 큰 원소가 가장 오른쪽에 위치하게 됩니다. (왜 그런지는 잘 알겠지요?)

(**1** **4** 2 5 8) -> (**1** **4** 2 5 8)

이제 다시 처음부터 두 원소를 골라 비교합니다.

(1 **4** **2** 5 8) -> (1 **2** **4** 5 8)

위와 같은 작업들을 쭉 시행합니다.

(1 2 **4** **5** 8) -> (1 2 **4** **5** 8)

(1 2 4 **5** **8**) -> (1 2 4 **5** **8**)

마지막까지 비교하였다면 다시 처음으로 갑니다.

그렇다면 이를 언제까지 반복해야 할까요? 더이상 자리가 바뀌는 원소들이 없을 때 까지 해야 하겠죠?

(**1** **2** 4 5 8) -> (**1** **2** 4 5 8)

(1 **2** **4** 5 8) -> (1 **2** **4** 5 8)

(1 2 **4** **5** 8) -> (1 2 **4** **5** 8)

(1 2 4 **5** **8**) -> (1 2 4 **5** **8**)

위 작업을 완료하였다면, 컴퓨터는 더이상 자리가 바뀌는 원소들이 없다는 것을 알아채고 정렬을 그만하게 됩니다. 음, 역시 정확하게 `1,2,4,5,8` 로 정렬이 되었군요.

일단, 위 버블 정렬 알고리즘을 C 언어에서 구현하기 위해서 저는 여러분들이 다음과 같은 함수를 만들어주기를 원합니다.

```
Bubble_sort(int* arr, int num_elements), swap(int* pele)
```

`Bubble_sort` 함수는 말그대로 정렬을 하는 함수 입니다. 이 때, `num_elements` 로 `arr` 이 가리키는 배열의 원소 개수를 알아야 하겠죠? 그리고 `swap` 함수는 `pele` 가 가리키는 원소와 그 다음 원소를 서로 뒤바꿔주는 함수 입니다. 따라서 `Bubble_sort` 함수가 `pele` 함수를 호출해야 되겠죠?

사실 버블 정렬은 매우 비효율적인 정렬 알고리즘 입니다. 하지만 구현하가 매우 단순하여 정렬해야 될 것이 작은 경우에는 이를 자주 이용하게 되지요. 정렬 알고리즘에 대해 궁금하신 분들은 여기를 클릭해서 정렬 알고리즘의 세상에 빠져보세요.

생각해볼 문제 2

유클리도 호제법을 이용해서 N 개의 수들의 최대공약수를 구하는 함수를 만들어보세요. 유클리드 호제법이 무엇인지 모르신다면, 인터넷 검색을 활용하는것을 추천합니다. (댓글을 달아도 돼요)

유클리드 호제법은 어떠한 두 수의 최대공약수를 계산하는데 쓰이는 방법입니다. 방법 자체는 간단합니다.

1. 두 수를 `m` 과 `n` 이라 하자. (`m > n`)

2. `m` 을 `n` 으로 나눈 나머지를 계산한다. (`m % n`)
3. `m % n` 이 0 이라면 `n` 값이 맨 처음 두 수의 최대공약수이다. (종료)
4. `m % n` 이 0 이 아니였다면, `m % n` 과 `n` 중 큰 것을 `m`, 작은 것을 `n` 이라 한 후 ① 로 돌아간다.

예를 들어서 63 와 35 의 최대공약수를 구한다고 합시다. 그렇다면 유클리드 호제법을 이용하면 아래와 같은 과정을 거칩니다.

1. `m = 63, n = 35`
2. `63 % 35 = 28`
3. 28 이 0 이 아니므로, 28 과 35 를 비교한느데 35 가 크므로 `m = 35, n = 28`
4. `m = 35, n = 28`
5. `35 % 28 = 7`
6. 7 이 0 이 아니므로, 7 과 28 을 비교, 28 이 크므로 `m = 28, n = 7`
7. `m = 28, n = 7`
8. `28 % 7 = 0`
9. 0 이므로, `n` 값 (7) 이 맨 처음 두 수의 최대공약수. 즉, 63 과 35 의 최대공약수는 7 이다

사실, 왜 위 과정을 거치면 두 수의 최대공약수가 나오는지에 대한 증명은 간단합니다. 수학적 지식이 없다면 이해가 안갈 수 도 있지만,

$$m = qn + r, (0 \leq r < q), \gcd(m, n) = \gcd(qn + r, n) = \gcd(r, n)$$

때문에 그렇습니다. 유클리드 호제법은 두 개의 수의 최대공약수를 찾는데에만 사용하였지만 이를 어떻게 `N` 개의 수의 공통된 최대공약수를 찾는데 응용할 수 있을까요? 답은 간단합니다. 처음 두 수의 최대 공약수를 구합시다. 그리고, 그 다음수와 구한 최대공약수의 최대 공약수를 계산합니다. 그리고 이를 쭈우욱 반복합니다.

예를 들어서 `18, 24, 40, 60` 의 최대공약수를 구한다고 해봅시다. 18 과 24 의 최대공약수는 6 입니다. 그러면 이제 6 과 40 의 최대공약수를 계산합니다. 이는 2 입니다. 그러면 이제 2 와 60 의 최대공약수를 계산합니다. 이는 2 입니다. 따라서, 이 4 개의 수의 공통된 최대공약수는 2 가 됩니다.

여기에도 수학적 원리가 있지만 간단하기 때문에 넘어가도록 하겠습니다. 여러분이 생각해보세요~

생각해볼 문제 4

자기 자신을 호출하는 함수를 이용해서 1 부터 특정한 수까지의 곱을 구하는 프로그램을 만들어 보세요.

여러분은 '자기 자신을 호출한다' 에 대해서 많은 고민을 많이 하셨을 것입니다. 일단, 아래의 코드를 직접 컴파일 후 실행해보세요

```c
#include <stdio.h>
int recursive(int n) {
  printf("난 인자가 %d 에요! \n", n);
  if (n <= 0) return 0;

  recursive(0);
}
int main() {
  recursive(3);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
난 인자가 3 에요!
난 인자가 0 에요!
```

흠. 어느 정도 예측 가능했던 결과입니다. 그렇다면, 아래의 코드는 어떨까요?

```c
#include <stdio.h>
int recursive(int n) {
  printf("난 인자가 %d 에요! \n", n);
  if (n <= 0) return 0;

  recursive(n - 1);

  return 0;
}
int main() {
  recursive(3);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
난 인자가 3 에요!
난 인자가 2 에요!
난 인자가 1 에요!
난 인자가 0 에요!
```

일단, 컴퓨터 상에서 위 코드는 아래의 순서로 실행됩니다.

1. `main` 함수에서 `recursive(3)` 을 호출함
2. `recursive` 에서 `n = 3` '난 인자가 3 이에요' 를 출력
3. `n` 이 0 이하가 아니므로 넘어감
4. `recursive(n-1)` 즉 `recursive(2)` 를 호출
5. `recursive` 에서 `n = 2` '난 ~ 2 ~ ' 를 출력 (~ 는 생략)
6. `n` 이 0 이하가 아니므로 넘어감
7. `recursive(n-1)` 즉 `recursive(1)` 을 호출
8. `recursive` 에서 `n = 1`, ' ~ 1 ~ ' 를 출력
9. `n` 이 0 이하가 아니므로 넘어감
10. `recursive(n-1)` 즉 `recursive(0)` 을 호출
11. `recursisve` 에서 `n = 0`, '~ 0 ~' 을 출력
12. `n` 이 0 이하이므로 `return 0;`
13. `n = 1` 이였던 `recursive` 에서 `return 0;`
14. `n = 2` 이였던 `recursive` 에서 `return 0;`
15. `n = 3` 이였던 `recursive` 에서 `return 0;`
16. `main` 함수로 돌아감.

사실, 위 작업이 이해가 잘 안되는 수도 있습니다만.. 나중에 변수의 정의 범위에 대해 배우게 된다면 좀더 쉽게 이해할 수 있으실 것입니다. 아무튼 위 사실을 활용해서 1 부터 `n` 까지 곱하는 재귀 함수를 만들어보세요 (위와 같이 자기 자신을 호출하는 함수를 **재귀함수**, 영어로 **recursive function** 이라 합니다.)

생각해볼 문제 4

계산기를 만들어보세요. 사용자가 1 을 누르면 +, 2 를 누르면 - 와 같은 방식으로 해서 만들면 됩니다. 물론 이전의 계산 결과는 계속 누적되어야 하고, 지우기 기능도 있어야 합니다. (물론하나의 함수에 구현하는 것이 아니라 여러개의 함수로 분할해서 만들어야겠죠?)

이 문제는 그다지 어려운 아이디어 같은 것이 필요한 것이 아니므로 생략하도록 하겠습니다. 사실, 난이도는 중하 정도 됩니다.

생각해볼 문제 5

N 진법에서 M 진법으로 변환하는 프로그램을 만들어보세요. (난이도 : 中)

사실 이 문제는 잘못낸 문제입니다. 물론 아이디어는 충분히 구현할 수 있지만 여러분은 아직 '문자열'에 대한 개념이 없기 때문에 정확하게 구현할 수 는 없지만 생각 정도는 할 수 있습니다. 일단 위 프로그램을 어떻게 만들 것인지에 대해 생각해 놓은 것을 보세요. 나중에 필요한 개념을 다 배우고 나면 하실 수 있을 것입니다. (참고적으로 문제에 조건 하나가 빠졌는데 `N,M` 은 모두 36 이하 입니다. 왜냐하면 숫자를 이용시 `0,1,...,9,A,B,..` 로 사용하는데 알파벳이 26 개이므로 총 36 진수까지 나타낼 수 있거든요)

- 사용자로 부터 무슨 진법에서 무슨 진법으로 변환할 지 입력받습니다. (`N,M` 입력)
- `N` 진법의 수를 입력받습니다.
- 그 수를 각 자리로 분해해 `int` 배열에 값을 넣습니다. 이 때, 값은 십진수로 넣습니다. 예를 들어서 16 진법으로 `7AE` 를 입력받았다면 `digit[0] = 14, digit[1] = 10, digit[2] = 7` 로 넣으면 됩니다. 참고로, 올바르지 않은 숫자가 사용되면 종료합니다. (예를 들어서 2 진법인데 3 이란 숫자를 사용함)
- 이 수를 십진수로 변환합니다. (`NtoDec` 함수 제작 요망)
- 이 십진수를 다시 `M` 진법의 수로 변환합니다. (`DectoM` 함수 제작 요망)

물론, 이 문제는 꼭 안푸셔도 됩니다. 나중에 개념을 좀더 배우다 보면 풀 수 있는 스킬들을 습득하실 것입니다.

생각해볼 문제 6

에라토스테네스의 체를 이용해서 1 부터 N 까지의 소수를 구하는 프로그램을 만들어보세요. (난이도 : 中)

에라토스테네스의 체.. 이름이 참 어렵군요. 사실 이는 간단합니다. 말그대로 숫자들만 걸러내는 **체(sieve)** 인데, 아래와 같은 방식으로 숫자를 걸러내어 소수들을 찾습니다.

- 수들을 쭉 쓴다.
- 2 의 배수들을 다 지운다.
- 2 에서 가장 가까운 안지워진 수를 찾는다. 아마도 3 일 것이다. (소수 찾았다!)
- 3 의 배수들을 다 지운다.
- 3 에서 가장 가까운 안지워진 수를 찾는다. 아마도 5 일 것이다. (소수 찾았다!)
- 5 의 배수들을 다 지운다.
- 5 에서 가장 가까운 안지워진 수를 찾는다. 아마도 7 일 것이다. (소수 찾았다!)
- 7 의 배수들을 다 지운다.
- 그 뒤로 쭈우욱 같은 작업을 실시

위키피디아에서 이 과정을 알기 싶게 애니메이션으로 나타낸 자료가 있으니 보시기 바랍니다.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

생각해볼 문제 7

1000 자리의 수들의 덧셈, 뺄셈, 곱셈, 나눗셈을 수행하는 프로그램을 만들어보세요. 나눗셈의 경우 소수 부분을잘라버리세요. 물론, 소수 부도 1000 자리로 구현해도 됩니다. 1000 자리 수들의 연산 수행 시간은 1 초 미만이여야합니다. (난이도 : 上)

`int` 자료형은 대략 42 억, 그러니까 10 자리 정도의 수 밖에 사용할 수 없었습니다. 그런데 문제에서 요구하는 것은 무려 1000 자리나! 이걸 도대체 어떻게 하라는 말일까요. 사실, 간단합니다. 크기가 1000 인 `char` 배열을 만들어서 배열의 한 원소를 수의 한 자리라고 생각하면 되죠. 이게 도대체 무슨 말이냐고요?

예를 들어서 `char BigNum[1000];` 을 정의하였다고 할 때, 사용자가 만일 123456 을 입력하였다면 `BigNum[999] = 6, BigNum[998] = 5, ... BigNum[994] = 1` 로 하면 되죠. 이러한 형식의 두 수를 더하는 연산을 하기 위해서는 각 원소를 더한 뒤, 받아올림이 있으면 그 다음 원소에 더해주고 하는 방식으로 쭉 나가면 됩니다. 어때요? 간단하죠.

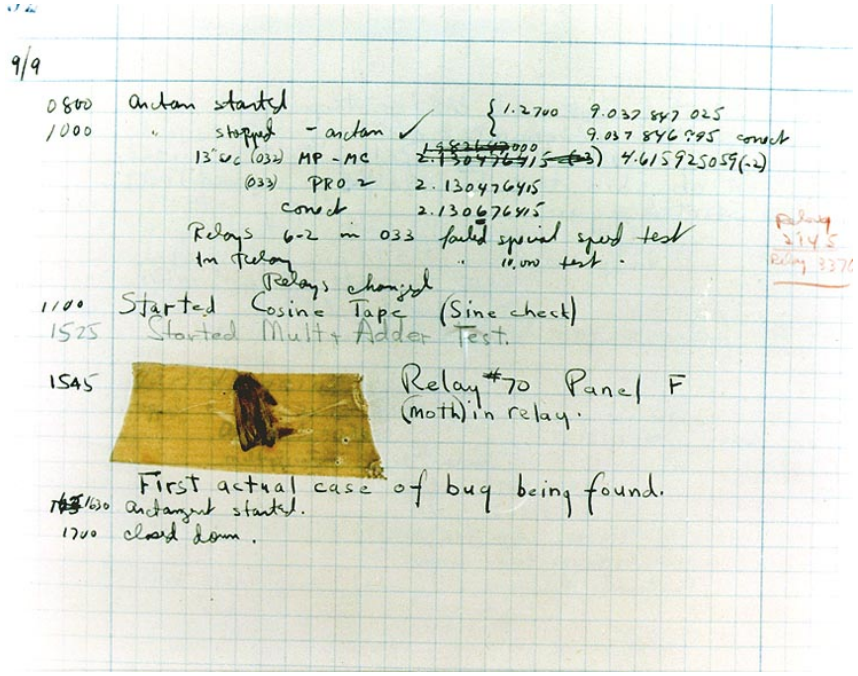
곱셈은 쉽게 하면 덧셈을 여러번 반복해서 호출하는 것으로 해결될 수 있지만 연산 속도가 느리므로 인간이 곱셈하는 방식으로 하는 것을 권합니다. 그렇다면 문제는 나눗셈인데 나눗셈 역시 뺄셈을 반복하는 것으로 해결 될 수 있지만 역시 느리므로, 인간이 나눗셈 하는 방식으로 계산하는 함수를 만들어보세요. 그럼. 행운을 빕니다.

12

디버깅(Debugging)

우리는 흔히 컴퓨터에 오류가 생기면 **버그(bug)** 가 생겼다고 합니다. 그런데 왜 하필이면 버그일까요?

곤충? 곤충이 뭐 어째서 말이지요. 사실 이 말이 나온 계기는 먼 옛날 1940 년대로 돌아갑니다. 유명한 여자 컴퓨터 과학자였던 **그레이스 호퍼(Grace Hopper)** 는 하버드 대학교의 Mark II 컴퓨터를 작동시키던 중 연산에 문제가 생기는 바람에 원인을 분석하다가 컴퓨터에 나방이 들어가 일으켰다는 사실을 알게되었습니다.[1] . 호퍼는 이 나방을 꺼내고는 곤충을 잡았다 해서 디버그(Debug) 했다고 기록했습니다.



이 일을 이후로 컴퓨터에 발생한 문제는 버그라고 하게 되었고, 이를 고치는 일은 **디버그** 라고 부르

1) 당시 컴퓨터는 지금의 쬐그만한 컴퓨터가 아니라 거의 학교 교실 만한 수 톤짜리 덩치였습니다. 현대의 개인용 컴퓨터에 나방이 들어가는 것은 이상한 일이지만 옛날에는 충분히 있을만한 일이였지요

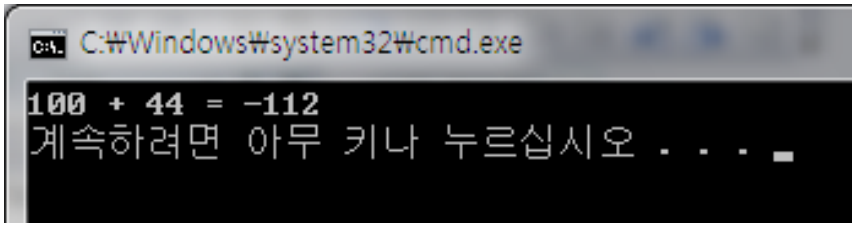
게 되었습니다. 참고적으로 이야기 하자면 기술적인 결함을 지칭하는 용어 버그는 호퍼 보다 훨씬 이전인 에디슨이 가장 먼저 사용하였습니다. 하지만 실질적으로 컴퓨터에서 버그가 사용된 것은 그레이스 호퍼가 처음이였지요.

우리는 이렇게 프로그램을 짜다 보면은 버그를 만나는 일이 종종 있습니다. 이번 강좌에서는 우리가 만든 프로그램의 버그를 찾아내는데 큰 도움을 주는 비주얼 스튜디오의 디버깅을 이용해보도록 하겠습니다. 뜬금없이 C 언어 강좌에 왜 이것을 갑자기 끼워넣었냐면 이 디버깅은 많은 C 언어 책들이 다루고 있는 내용은 아니지만 여러분이 C 프로그래밍을 배우다 보면 꼭 필요한 스킬이기 때문입니다. 적절한 디버깅을 통해서 여러분의 프로그램의 골치아픈 문제점들을 찾아낼 수 있습니다.

먼저 `Visual Express 2008` 을 실행하셔서 다음의 코드를 복사해 넣어 봅시다.

```c
#include <stdio.h>
int main() {
  char a, b, c;
  a = 100;
  b = 300;
  c = a + b;

  printf("%d + %d = %d \n", a, b, c);
  return 0;
}
```

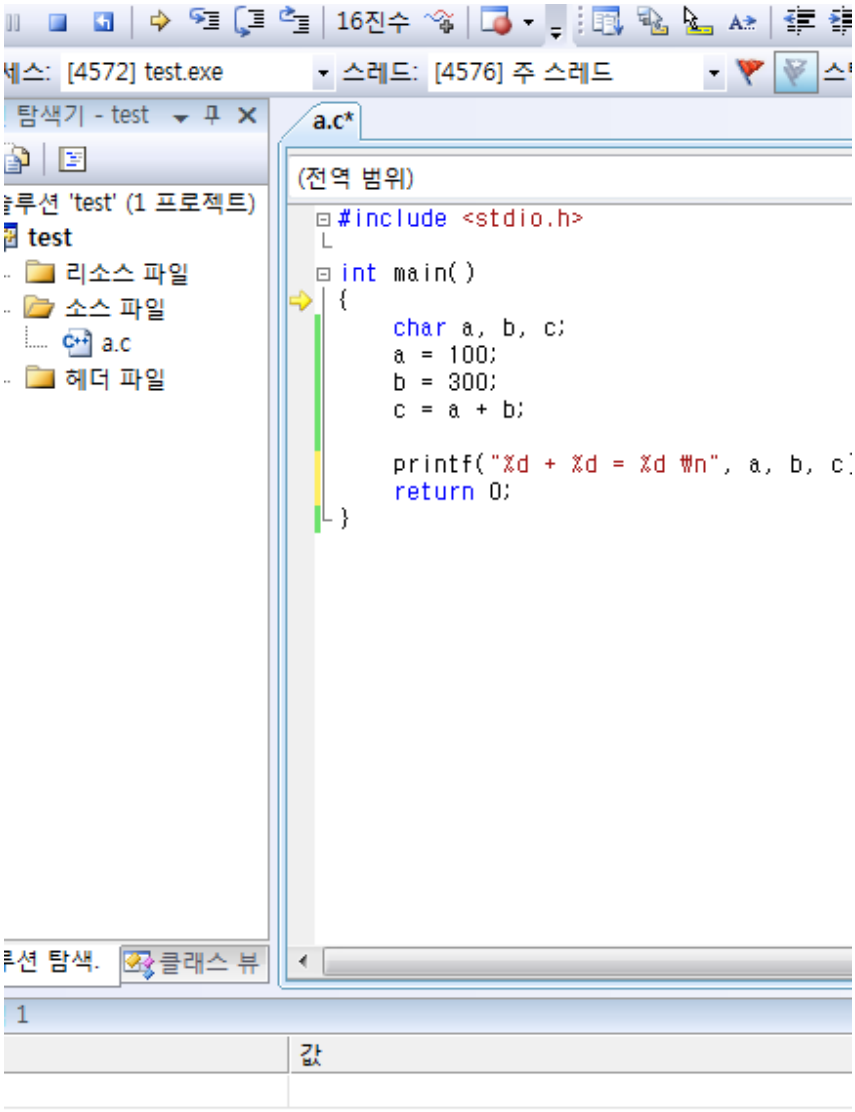
성공적으로 컴파일 하면 아래의 화면을 볼 수 있습니다.



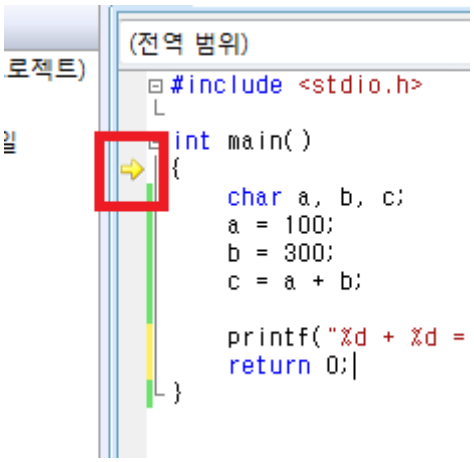
으응? 우리는 흠칫 놀랄 수 밖에 없습니다.내가 b 에 300 이라는 값을 주었는데 위에 출력된 결과는 44 로 나왔습니다.게다가 44 는 그렇다 치고, 양수 + 양수를 했는데 음수가 나오다니요. 컴퓨터가 계산을 잘못한 것일까요? 음.. 아무리 생각해도 그럴 일은 있을 수 없겠군요. 그렇다면 우리는 여기서 두 가지 방법으로 왜 틀린 결과가 나왔는지 찾아낼 수 있습니다.

첫번째 방법은 밤새도록 머리를 쥐어 짜고 도대체 왜 저딴 결과가 나왔는지 고민하는 것입니다. 아무래도 일찍 자기 위해서는 효과적인 방법이 아니지요. 하지만 다행이도 두번째 방법이 있습니다. 바로, '컴퓨터의 관점에서 코드를 따라가 보는 것' 입니다. 말이 조금 어렵지만, 이 과정이 바로 앞서 이야기 한 '디버그' 입니다.

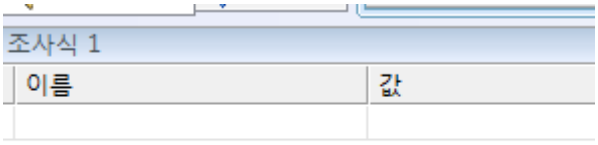
일단, `Visual Express` 에서 `F10` 을 눌러 봅시다.

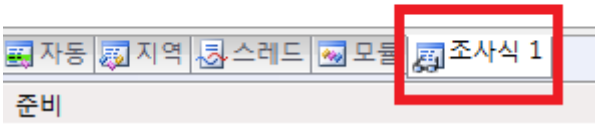


오오. 무언가 새로운 창이 떳습니다. 일단, 메뉴바 쪽에 알 수 없는 내용들은 무시하고 중점적으로 살펴볼 것은 아래와같습니다. 붉은색 박스로 친 노란색의 화살표와,



맨 아래쪽의 '조사식' 부분을 살펴봅시다.



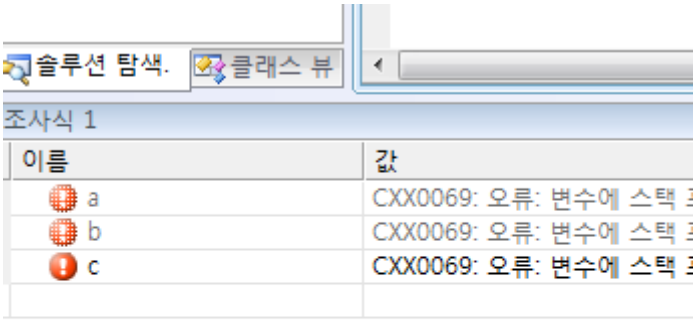


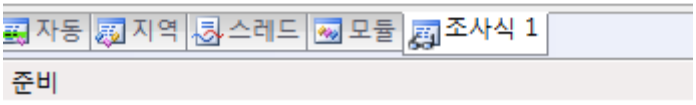
앞에서도 말했듯이 우리가 디버깅을 하는 이유는 버그를 찾아 내기 위해서입니다. 그런데, 우리가 쉽게 버그를 찾지 못하는 이유는 바로 컴퓨터가 눈 깜짝할 사이에 명령을 다 실행해 버리기 때문이죠. 만일 우리가 컴퓨터가 수를 더하고 출력하는 과정을 천천히 눈으로 볼 수 있다고 치면, 굳이 버그를 쉽게 찾을 수 있습니다. 디버깅을 하면 사용자로 하여금 각 문장이 실행되는 과정을 천천히 살펴 볼 수 있게 해서 어느 문장에서 문제가 발생하는지 알 수 있게 해줍니다. 즉, 우리가 프로그램을 실행해 버리면 컴퓨터는 문장을 순식간에 다 실행해 버리지만 디버깅을 통해 사용자가 순차적으로 문장을 하나씩 하나씩 실행할 수 있게 해주는 것입니다.

위 말이 이해가 잘 안되도 직접 해보면 쉽게 느낄 수 있을 것입니다.

먼저, 노란색 화살표 부터 살펴봅시다. 이는 '내가 다음에 실행한 코드' 를 가리키는 역할을 합니다.

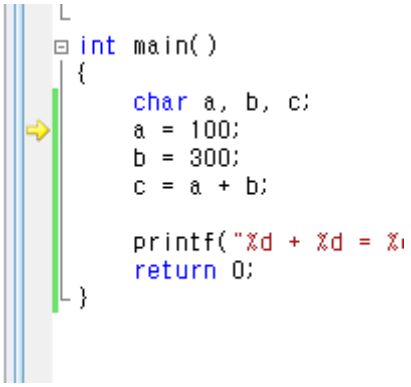
아래 조사식은 내가 값을 보고 싶은 식을 써 넣으면 됩니다. 예를 들어서 내가 변수 a 의 값을 보고 싶다면 a 를 치고, a + b 의 값을 보고싶다면 a + b 의 값을 치면 됩니다. 이 디버깅 과정에서 문제가 생기는 것은 b, c 이므로, 일단 변수 a,b,c 의 값들이 어떻게 변화하는지 살펴보도록 합시다.



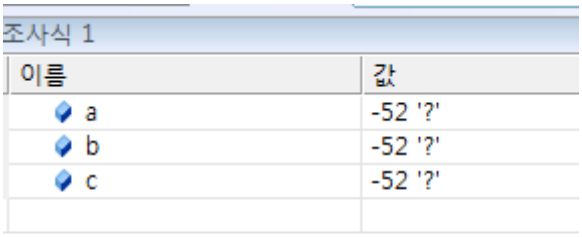


위와 같이 a,b,c 를 차례로 입력합니다. 어랏. 이상합니다. 변수 a,b,c 의 값이 출력되고 있지 않습니다. 왜 그럴까요? 사실, 당연한 일이지요. 노란색 화살표를 보면 현재 { 를 가리키고 있었습니다. 다시 말해 다음에 실행할 문장이 { 이므로 char a,b,c 는 실행 조차 되지 않았기 때문이죠. 따라서 컴퓨터는 a,b,c 라는 변수가 정의되어 있는지 모르기 때문에 위와 같이 a,b,c 라는 변수가 없다는 오류를 내게 됩니다.

그 다음 문장을 실행하기 위해서는 F10 을 누르면 됩니다.

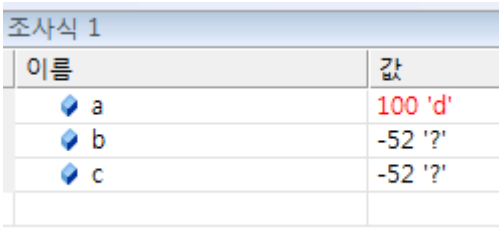


어랏. 분명히 그 다음 문장을 실행한다고 그랬는데 두 줄이나 내려왔습니다. 이는 사실, 컴퓨터에서 사용자의 편의를 위해 변수를 정의하는 부분은 일일이 귀찮게 F10 을 누르지 않도록 자동으로 실행해 준 것입니다. 아무튼. 크게 중요한 부분은 아닙니다. 다음 문장을 실행하기 전에, 조사식이 어떻게 되었는지 봅시다.



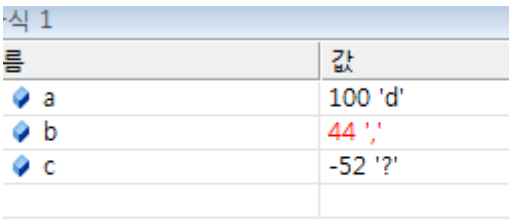
모두 -52 라는 값을 갖고 있는 것처럼 보입니다. 사실, 그렇지 않습니다. 우리가 a,b,c 에 아직 아무런 값을 대입하지 않았기 때문에 현재 쓰레기 값으로 초기화 된 것입니다. 즉, -52 라는 것은 아무런 의미가 없습니다. 단순히 a,b,c 에 아무런 값도 대입되지 않았음을 나타냅니다.

이제, F10 을 또 한번 눌러봅시다. 화살표가 b=300; 을 가리켰으므로, 그 위의 문장, 즉 a = 100; 이란 문장을 실행했다는 것입니다. 따라서, 조사식을 보면,

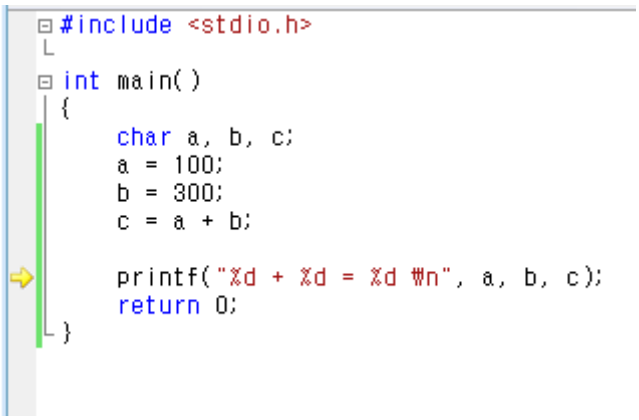


위와 같이 a 의 값이 100 이라고 나옵니다. 이 때 옆의 'd' 는 100 에 해당하는 아스키 문자(기억하시죠? 모르면 5강 참조) 을 나타낸 것으로, 사용자의 편의를 위해 컴퓨터가 나타내주었습니다.

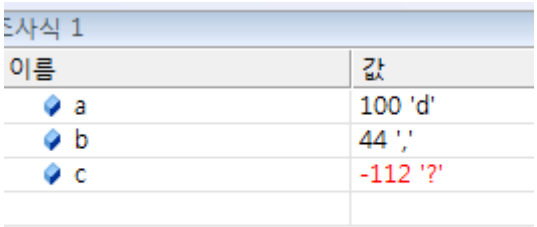
마찬가지로 F10 을 눌러서 b 의 값도 정의해줍시다.



허걱. 분명히 b = 300 을 했는데 b 에는 44 가 들어갔습니다. 무언가 문제가 있어 보입니다. 사실, 이쯤 되면 무엇이 문제인지 짐작할 수 있겠지만, 아래에서 설명하도록 하죠. 마찬가지로 F10 을 또 한번 눌러서 c = a + b; 를 실행해 보면 아래와 같습니다.



조사식을 살펴 보면



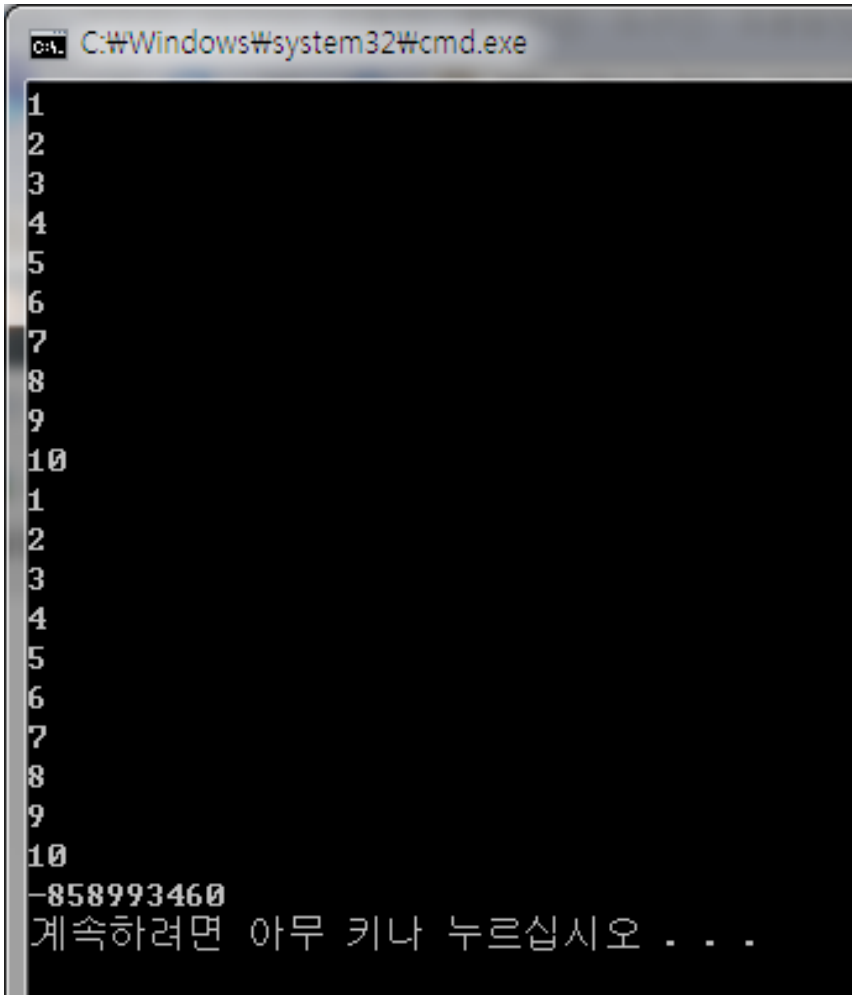
c 의 값이 -112 로 되었음을 볼 수 있습니다. 또한, -112 에 해당하는 아스키문자가 ? 여서 '?' 로 출력된 것이 아니라 아스키 표에 해당하지 않는 수이기 때문에 알 수 없음의 의미로 ? 가 출력된 것입니다. 음. `c = a + b;` 를 했는데 왜 c 에 -112 가 들어갔을까요? 사실, 여러분이 다 짐작하고 있겠지만 이유는 간단합니다. char 의 범위가 128 까지 이기 때문이죠. 즉, b 값에 44 가 들어간 것도, `100 + 44` 를 했는데 -112 가 출력된 것도 모두 char 의 범위가 128 까지 이기 때문에 발생한 일들입니다.

만일 우리가 디버깅을 하지 않았다고 칩시다. 과연 우리는 char 의 범위 때문에 그렇다라는 것을 알 수 있었을까요? 물론, 위 예제에선 그렇습니다. 예제가 간단하기 때문이죠. 우리는 손쉽게 내가 char 형의 범위를 무시하고 값을 대입해서 오류가 떴구나 라는 사실을 알 수 있습니다. 그러나, 실제 예제는 이와 같이 단순하지 않습니다. 위는 그냥 설명하기 편하게 하기 위해 위와 같이 예를 잡은 것이고, 실제 우리가 만들게 될 프로그램은 이것보다 100 배는 더 복잡합니다.

예를 들어 아래의 코드를 봅시다(사실 아래 것도 그리 복잡한 것은 아니지만 적절한 예제가 없어서.. 여러분이 직접 프로그래밍 하시다 보면 알게 될것입니다) .

```c
/* 샘플 코드 */
#include <stdio.h>

int main() {
  int arr[10];
  int i;

  for (i = 0; i < 10; i++) {
    scanf("%d", &arr[i]);
  }
  for (i = 0; i <= 10; i++) {
    printf("%d \n", arr[i]);
  }
  return 0;
}
```

성공적으로 컴파일 해보면 아래와 같이 맨 아래에 이상한 값이 출력됨을 볼 수 있습니다.



도대체 -858993460 은 어디서 나온 것일까요? 한 번 여러분이 디버깅을 통해 코드의 어느 부분이 잘못되었는지 찾아서 수정해보세요 :)

13

문자열 (string)

안녕하세요. 벌써 15 강에 도달하였습니다. 정말로, 거대한 숲을 거침없이 헤쳐왔다는 느낌이 듭니다. 제가, 첫 강좌를 올린 것이 2009년 4월 16일 이였는데, 벌써 2009년의 마지막에 다다르고(제가 이 글을 처음 시작했을 때만)있습니다. 인터넷을 뒤져보면 많은 C 언어 강좌들이 있는데, 유료 강좌 빼고는 제 강좌처럼 근성 있게 올라오는 것도 드문 것 같네요. 암튼, 저나 여러분이나 정말로 대단한 사람들 입니다 ㅎㅎ.

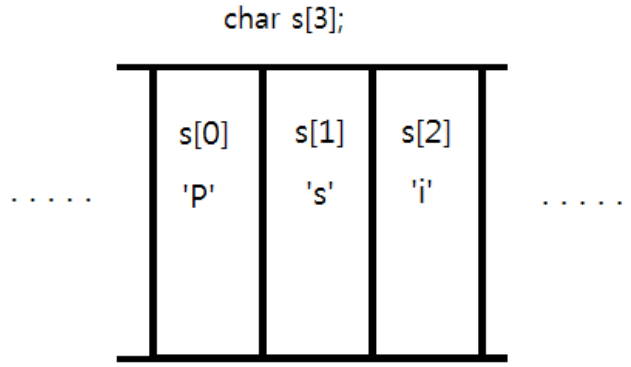
그 동안, 따분한 숫자들만 가지고 놀아서 조금 지루한 면이 적지 않아 있었습니다. 그래서, 이번 강좌에서는 문자들, 말그대로 문자의 나열인 **문자열(string)**에 대해서 이야기 해보도록 하겠습니다.

문자열은 영어로 `string` 이라고 하는데, 원래의 의미는 실 입니다. 그런데, 문자열을 `string` 이라 부르는 이유는 정말 문자열이 실 처럼 문자들이 쭈르륵 나열 된 것이기 때문이죠. 참고적으로 문자열에 대해 지금 처음 배우시는 분들은 5 강을 다시 한 번 읽어보시기 바랍니다. 아무래도 예전에 배운 내용이라 까먹었을 확률이 매우 높거든요.

그렇다면, 컴퓨터는 문자열을 어떻게 저장할까요. 앞서, 제가 쭈르륵 나열 되어 있다는 사실을 강조했다는 부분을 생각해보면, 문자열을 문자들의 배열, 즉 `char` 배열에 저장함을 알 수 있습니다.

널 - 종료 문자열 (Null-terminated string)

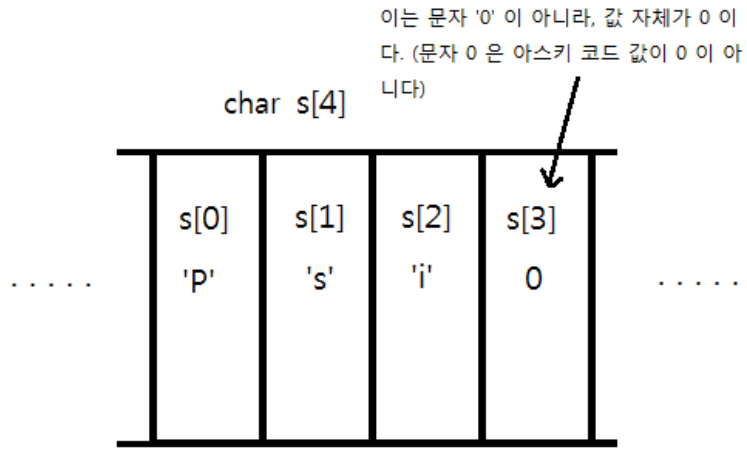
위 소제목이 무슨 뜻인지 모른다고 해서 겁을 먹으신 분들이 있을텐데, 조금 있다가 알게 될 것입니다. 앞서, 컴퓨터에서는 문자열을 `char` 배열에 저장한다고 하였습니다. 다시말해, 아래 그림과 같이요.



물론, 실질적으로 메모리에 저장된 값은 각 문자에 해당하는 아스키 값이다. 이를 문자로 생각하냐, 숫자로 생각하냐는 프로그래머 마음이다.

근데, 말이죠. 위와 같이 문자열 s 를 정의하였을 때, 무언가가 불편할 것 같지 않나요? 만일, 우리가 char 배열 s 에 저장된 문자들을 화면에 출력한다고 해봅시다. 이상적인 상황으로는 컴퓨터에게 "s 의 문자열을 출력해" 라고 말하면 알아서 출력해주는 것입니다. 그러나, 위와 같이 배열 s 에 문자를 저장하면 "s 의 문자열을 출력해. 근데, 그 문자가 아마 3 문자일거야" 라고 말해주어야 하는 불편함이 생긴다는 말입니다.

문자열은 말그대로 문자들이 하나로 뭉쳐서 다니는 것이기 때문에 (만일 우리가 s 의 문자열을 이용한다고 하면 첫글자 P 만 이용할 것입니까? 아니죠. 상식적으로 Psi 전체를 하나로 이용하는 것이죠), 문자열을 이용할 때 마다 문자열의 길이를 알아야 한다면 정말로 불편한 일이 아닐 수 없습니다. 그래서, C 개발자들은 아래와 같이 멋진 대안을 내놓았습니다.



물론, 실질적으로 메모리에 저장된 값은 각 문자에 해당하는 아스키 값이다. 이를 문자로 생각하냐, 숫자로 생각하냐는 프로그래머 마음이다.

위와 같이 문자열의 끝에, **여기 까지가 문자열이였습니다** 라고 알려주는 종료 문자를 넣은 것입니다. 이 종료 문자는 아스키 값이 0 이고, '\0' 라고도 나타냅니다. 절대 문자 '0' 하고 헷갈리면 안됩니다. 문자 0 은 아스키 코드 값이 0 이 아니라 48 입니다. 흔히, 이 종료 문자를 가리켜서 **널(Null)** 이라고 부릅니다. 이제, 널 종료 문자라는 말의 의미를 알겠죠? 말그대로, 널로 끝나는 문자 라는 의미입니다. 이것이 바로, C 언어의 문자열의 기본적인 형태 입니다.

널 문자가 들어갈 공간이 있어야 하기 때문에 3 글자라고 해도, 배열은 4 칸이 필요하게 됩니다. 위와 같이 s[4] 처럼요. 그럼, 위와 같이 널 종료 문자가 편리한 이유는 컴퓨터가 문자열의 끝을 쉽게 구할 수 있기 때문입니다. 우리가 굳이 '이 s 문자열은 3 문자인데, 출력해죠' 라고 말할 필요 없이, 's 문자열을 출력해' 란 말만 해주어도 컴퓨터가 알아서 '음, 널이 나올때 까지 출력해야지' 라고 출력한다는 것입니다.

```c
/* 널 뽀개기 */
#include <stdio.h>

int main() {
  char null_1 = '\0';  // 이 3 개는 모두 동일하다
  char null_2 = 0;
  char null_3 = (char)NULL;  // 모두 대문자로 써야 한다

  char not_null = '0';

  printf("NULL 의 정수(아스키)값 : %d, %d, %d \n", null_1, null_2, null_3);
  printf("'0' 의 정수(아스키)값 : %d \n", not_null);

  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
NULL 의 정수(아스키)값 : 0, 0, 0
'0' 의 정수(아스키)값 : 48
```

와우. NULL 의 정수값은 모두 0 이 출력되었고, 문자 '0' 의 정수값은 48 이 출력되었습니다.

```
char null_1 = '\0';  // 이 3 개는 모두 동일하다
char null_2 = 0;
char null_3 = (char)NULL;
```

위 세개의 문장의 각 char 변수에는 모두 동일한 값, 즉 0 이 들어가게 됩니다. null_1 의 경우 \0 의 값, 즉 \0 의 아스키 값이 들어가는데 \0 의 아스키 값은 0 입니다. 왜 '0' 이라 안쓰고 \0 이라 쓰는지는 알겠죠? '0' (문자 0) 의 아스키값은 48 이기 때문입니다.

(위에서 확인할 수 있듯이) 마찬가지로 null_2 에는 0 이 들어가고, null_3 에는 NULL 의 값이 들어가는데, NULL 은 0 이라고 정의되어 있는 상수 입니다. 따라서, null_3 에도 0 이 들어갑니다.

```
char not_null = '0';
```

반면의 not_null 의 경우 문자 '0' 의 아스키값이 들어가는데, 문자 0 의 아스키값은 48 입니다. 따라서, not_null 의 정수값을 출력할 때 48 이 출력되었습니다. 위 예제를 통해 우리는 문자열의 마지막에는 종료 문자로 '\0' 이나 NULL 혹은 문자 0 이 아닌 0 이란 값 자체를 사용할 수 있음을 알았습니다.

컴파일 오류

```
warning C4047: '초기화 중' : 'char'의 간접 참조 수준이 'void *'과(와)
↪  다릅니다.
```

참고적으로, 컴파일 시 위와 같은 경고를 만나는 분들이 있을 것입니다. 대다수의 경우 경고는 중요한 역할을 하지만 여기서의 경고는 별로 중요하지 않으니 상관하지 않으셔도 됩니다. 나중에, 위 경고가 왜 나왔는지 이야기 해보죠.

아래 예제를 통해 확실히 알아보죠.

```
/* 문자열의 시작 */
#include <stdio.h>
int main() {
```

```
  char sentence_1[4] = {'P', 's', 'i', '\0'};
  char sentence_2[4] = {'P', 's', 'i', 0};
  char sentence_3[4] = {'P', 's', 'i', (char)NULL};
  char sentence_4[4] = {"Psi"};

  printf("sentence_1 : %s \n", sentence_1);  // %s 를 통해서 문자열을 출력한다.
  printf("sentence_2 : %s \n", sentence_2);
  printf("sentence_3 : %s \n", sentence_3);
  printf("sentence_4 : %s \n", sentence_4);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
sentence_1 : Psi
sentence_2 : Psi
sentence_3 : Psi
sentence_4 : Psi
```

오오오. 모두 Psi 가 성공적으로 출력되었습니다. 일단, 각 문자열을 정의하는 것 부터 살펴보도록 합시다.

```
char sentence_1[4] = {'P', 's', 'i',
                      '\0'};  // \0 는 아스키값이 0 인 문자, 즉 종료 문자이다
```

일단, 첫번째 형식. 위는 sentence_1 이라는 크기가 4 인 char 배열을 정의하는 문장입니다. 각 원소에 차례로 'P', 's', 'i' 가 들어가고 그 뒤에 종료 문자 '\0' 이 들어갔습니다. 즉, sentence_1 은 완벽한 널 종료 문자열입니다. 마찬가지로 생각하면 sentence_2, sentence_3 도 널 종료 문자열 임을 알 수 있습니다. 그렇다면 sentence_4 의 정의 부분을 봅시다.

```
char sentence_4[4] = {"Psi"};
```

음. 이전까지 보아왔던 정의 형태 보다 약간 다르군요. 사실, 각 문자를 작은 따옴표로 표시해서 배열에 저장하는 일은 매우 번거로운 일이 아닐 수 없습니다. 그래서 C 언어에서는 위와 같이 문자들을 쭉 나열한 것을 큰 따옴표로 묶어주게 되면 알아서 각각의 문자로 넣어줍니다.

이 때, 널 문자는 뒤에 자동으로 추가 되니 굳이 큰따옴표 안에 특별히 명시해줄 필요는 없습니다.

초보자들이 흔히 하는 실수가 위와 같이 "Psi" 로 정의해놓고 배열의 크기를 3 으로 잡는 사람들이 간혹 있습니다. 이렇게 되면 sentence 에는 끝에 NULL 이 들어가지 않으므로 sentence 의 문자열을 출력하라고 했을 때 NULL 이 언제 나올지 모르기 때문에 혀용되지 않는 메모리 범위를 읽게되는 문제가 발생합니다.

반드시 널 문자를 위한 공간 하나를 더 추가하는 것을 잊지 마시기 바랍니다.

아무튼, 위와 같이 정의하면 이전의 sentence_1~3 과 정확히 동일한 문자열이 됩니다.

```
printf("sentence_4 : %s \n", sentence_4);
```

위는 'sentence_4 부터 들어 있는(sentence_4 는 배열의 시작점을 가리키고 있다는 사실을 알고 있겠죠?) 문자열을 출력해달라' 라는 의미로 %s 를 이용하였습니다. 이전의 %c 는 한 문자만을 출력하는 것이지만 %s 를 이용한다면 sentence_4 에서 부터 널이 나올 때 까지 문자를 계속 출력하게 됩니다.

따라서, 위와 같이 Psi 가 예쁘게 출력된 것입니다.

여기 까지 도달 하셨다면 약간 헷갈리는 것이 있을 수 있습니다. " " 와 ' ' 의 차이점은 뭐지? 말이죠. 저의 경우 이를 잘 이해하지 몰라서 많은 시간 애를 먹었는데 여기 여러분을 위해 "" 와 '' 의 차이점을 깔끔하게 정리한 표를 소개합니다.

""	''
큰따옴표는 문자열 (한 개 이상의 문자)를 지정할 때 사용된다.	작은 따옴표는 한 개의 문자를 지정할 때 사용된다.
예) "abd", "asdfasdf", "sentence", "a" 등등	예) 'a', 'b', '\0' (틀린표현 : 'abc', 'ab', 'cd' 등등)

무언가 깔끔하게 정리된 느낌이 드나요? 사실, 아직 들기는 힘듧니다. 다만, 이 강좌의 끝부분을 읽고 있을 때 쯤이면 그 차이가 완전히 머리속에서 정리 되기를 바랍니다.

```c
/* 포인터 간단 복습 */
#include <stdio.h>
int main() {
  char word[30] = {"long sentence"};
  char *str = word;

  printf("%s \n", str);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
long sentence
```

위 예제는 사실 단순합니다. 우리가 이전에 배운 내용에 따르면 char* 을 이용해서 char 배열을 가리킬 수 있다고 하였습니다. 위는 이를 그대로 적용 시킨 것으로 str 이라는 char 을 가리키는 포인터가 배열 word 를 가리키고 있습니다. 따라서,

```
printf("%s \n", str);
```

에서 str 이 가리키는 것을 문자열로 출력 (즉, 널이 나올때 까지 출력) 해 위와 같이 long sentence 가 나오게 된 것입니다.

```
/* 문자열 바꾸기 */
#include <stdio.h>
int main() {
  char word[] = {"long sentence"};

  printf("조작 이전 : %s \n", word);

  word[0] = 'a';
  word[1] = 'b';
  word[2] = 'c';
  word[3] = 'd';

  printf("조작 이후 : %s \n", word);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
조작 이전 : long sentence
조작 이후 : abcd sentence
```

사실 위 과정도 매우 단순합니다. 일단, 첫번째로 문자열을 정의하는 부분부터 살펴봅시다.

```
char word[] = {"long sentence"};
```

오잉? 원소의 개수를 지정하는 부분에 아무런 숫자도 써있지 않습니다. 하지만, 여태까지 강좌를 열심히 보아왔던 분들에게는 별 이상하게 느껴지지 않을 것 입니다. 왜냐하면 [ ] 안을 빈칸으로 두었다는 뜻은 컴파일러가 알아서 원소의 수를 세어서 빈칸을 채워 넣으라는 뜻이지요.

따라서, 우리는 귀찮게 한글자 한글자 세어서 값을 써줄 필요 없이 단순히 빈칸으로 남겨 놓으기만 하면 됩니다. (물론 배열의 정확한 크기를 알아야 할 상황이 온다면 특별히 값을 명시해 주어야 겠지만)

```
word[0] = 'a';
word[1] = 'b';
word[2] = 'c';
word[3] = 'd';
```

위와 같이 word 배열의 첫 4 개의 원소를 각각 a,b,c,d 로 변경하였습니다. 따라서 아래 printf 문에서 long 부분이 abcd 로 변경된 모습을 볼 수 있게 됩니다.

문자의 개수를 세자

나중에 프로그래밍을 하다 보면 특정한 문자열에 들어 있는 문자의 개수를 세는 일이 많을 것 입니다. 이를 수행하는 함수를 만들어 봅시다. 먼저, 여러분들께서 아래의 코드를 보지 말고 한 번 직접 작성해 보세요.

```
#include <stdio.h>
int str_length(char *str);
int main() {
  char str[] = {"What is your name?"};

  printf("이 문자열의 길이 : %d \n", str_length(str));

  return 0;
}
int str_length(char *str) {
  int i = 0;
  while (str[i]) {
    i++;
  }

  return i;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
이 문자열의 길이 : 18
```

소스 코드가 머리에 잘 다가오면 좋겠지만 일단 중요한 부분만 설명하고자 합니다.

```
int str_length(char *str) {
  int i = 0;
  while (str[i]) {
    i++;
  }

  return i;
}
```

일단 우리가 만들게 될 함수 이름은 str_length 함수 입니다. 인자는 char 형을 가리키는 포인터 형태 이므로, char 배열을 취할 수 있음을 알 수 있습니다. 이전에 함수 강좌에서도 이야기 했지만 일차원 배열을 가리키는 포인터는 (그 배열의 형)* 이라고 했죠? 아무튼, str 을 통해 문자열 배열을 가리킬 수 있습니다.

```
while (str[i]) {
  i++;
}
```

일단 while 문의 조건 부분에는 str[i] 가 들어 있습니다. 이 말은 즉슨, str[i] 가 0 이 될 때까지 i 의 값을 계속 증가 시키겠다죠? 그런데 문자열에서 str[i] 가 0 이 되는 순간은 언제일까요. 바로 NULL 문자 일 때, 즉 문자열의 끝 부분에 도달하였을 때 0 이 되는 것입니다. 다시말해 while 문에서 str[i] 가 문자열의 끝 부분이 될 때 i 값의 증가를 멈춘다는 것이지요.

따라서 i 에는 맨 마지막의 NULL 문자를 제외한 나머지 문자들의 총 개수가 되는 것입니다.

문자열 입력받기

```
/* 문자열 입력 */
#include <stdio.h>
int main() {
  char words[30];

  printf("30 자 이내의 문자열을 입력해주세요! : ");
  scanf("%s", words);

  printf("문자열 : %s \n", words);
```

```
  return 0;
}
```

실행 결과

```
30 자 이내의 문자열을 입력해주세요! :  WhySoSerious?
문자열 : WhySoSerious?
```

이번에는 문자열을 입력 받는 방법에 대해 이야기 하고자 합니다. 이전에 5 강에서 문자를 입력받는 방법에 대해 이야기한 적이 있는데 기억이 나실련지요? 문자열을 입력 받는 것도 그다지 다를 바 없습니다.

```
char words[30];
```

일단 최대 29 글자 까지 저장할 수 있는 문자 배열 words 를 생성하였습니다. 왜 30 글자가 아니라 29 글자인지는 잘 알겠지요? 끝에 널이 들어가기 때문이죠.

```
printf("30 자 이내의 문자열을 입력해주세요! : ");
scanf("%s", words);
```

이제, scanf 를 통해서 문자열을 입력받습니다. 일단, 입력 받는 형식이 %s 입니다. 기존의 하나의 문자는 %c 였는데 , 문자열의 경우 %s 를 이용합니다. 또한, 두번째 인자에 words 를 써주었는데, 약간 이상합니다. 이전에 입력 받을 때 에는

```
char c;
scanf("%c", &c);
```

와 같이 & 를 이용해서 주소값을 전달하였는데 여기서는 & 를 붙이지 않았습니다. 하지만, 여태까지의 강좌를 잘 읽어보셨고 특히 함수에 대해 잘 공부하신 분이라면 별 이상한 점을 못느꼈을 것입니다. 왜냐하면 words 라는 배열의 이름 자체가 배열을 가리키고 있는 포인터 이기 때문에 words 의 값을 전달함으로써 배열의 (시작) 주소값을 잘 전달할 수 있습니다.

scanf 함수는 잘 아시다싶이 엔터가 나올 때 까지 입력을 받습니다. 그런데 말이죠. 우리가 문자열을 적는데 띄어쓰기를 한다면 아래와 같이 이상한 일이 발생한다는 사실을 알 수 있습니다.

실행 결과

```
30 자 이내의 문자열을 입력해주세요! : what is your name
문자열 : what
```

분명히 scanf 는 엔터가 나오면 입력을 종료하는데 왜 what is your name? 에서 what 부분만 입력이 되었냐죠. 사실 이 부분에 대해서 설명하면 이번 강좌가 너무너무 길어지기 때문에 다음 강좌로 미루도록 합시다. 아무튼. 다음 강좌가 나올 때 까지 인터넷으로 조사 좀 해보세요. 그럼 이번 강좌는 여기에서 마칩니다.

생각해보기

문제 1

놀랍게도 배열을 할당하지 않고도 다음과 같이 문자열을 지정할 수 있습니다.

```c
char *str = "abcdefghi";
printf("%s", str); /* 하면 잘 출력된다.*/
```

그렇다면 위 str 과 char c_str[]={"abcdefghi"}; 의 차이점은 무엇일까요? (난이도 : 上)

문제 2

다음 문장이 왜 성립하지 않는지 생각해보세요 (난이도 : 中上)

```c
char str_a[] = "abc";
char str_b[] = "abc";

if (str_a == str_b) {
  /* 이 부분이 실행되지 않는다.*/
}
```

문제 3

(2) 의 답을 얻었다면 두 개의 문자열을 비교하는 함수를 만들어서 같으면 1, 다르면 0 을 리턴하게 해보세요. (난이도 : 中)

버퍼에 대한 이해

안녕하세요 여러분. 요즘에 제가 많이 바빠서 글을 자주 못올리고 있지만 여러분은 너그러운 마음으로 이해해 주시라 믿고 있습니다. 그렇다면, 지난번의 강좌를 계속 이어 나가도록 하겠습니다.

```c
/* 이상한 scanf */
#include <stdio.h>
int main() {
  int num;
  char c;

  printf("숫자를 입력하세요 : ");
  scanf("%d", &num);

  printf("문자를 입력하세요 : ");
  scanf("%c", &c);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
숫자를 입력하세요 : 1
문자를 입력하세요 :
```

허거걱! 여러분은 위 소스를 실행했을 때 충격을 금치 못했을 것입니다. 분명히 우리는 다음과 같이 scanf 를 이용해서 정수를 입력 받고 그 다음에 문자를 입력받으라고 명시했습니다.

```c
printf("숫자를 입력하세요 : ");
scanf("%d", &num);

printf("문자를 입력하세요 : ");
scanf("%c", &c);
```

분명히 우리는 컴퓨터로 하여금 숫자를 입력받은 후, "문자를 입력하세요 : " 를 출력한 다음, c 에 문자를 입력받으라고 명령하였습니다. 그리고, 컴퓨터가 정말 특별히 우리를 싫어하지 않는 한 무조건 우리의 말을 따라야 하는 것이지요. 그런데, 이게 무슨 일입니까? 우리의 컴퓨터는 우리가 친절히 명시해 준 scanf("%c", &c); 명령을 완전히 무시한 것 아닙니까? 이게 도대체 무슨 일이죠?? 사실, 우리가 컴퓨터에게 화풀이를 하기 전에 scanf 함수가 어떻게 작동하는 것인지 먼저 확인해볼 필요가 있습니다.

우리가 컴퓨터에 무언가를 입력하면 컴퓨터는 어떻게 처리를 할까요? 예를 들어서 우리가 컴퓨터에게 abcde 를 입력하였을 때, 컴퓨터가 각 문자를 입력받을 때 마다 처리를 한다면 (즉 우리가 a 를 누르는 순간 a 라는 문자를 변수에 저장하고 등등 작업을 하고 그 다음에 b 가 들어오면 다시 이 문자를 ...) 비효율적일 것입니다.

하지만 이렇게 하면 어떨까요. 우리가 문자를 입력한다면 다른 곳에 잠시 보관해 놓았다가 우리의 입력이 끝난다면 잠깐 보관해 놓았던 곳의 정보를 한꺼번에 처리하는 것입니다. 따라서, 만일 우리가 abcde 를 입력하였다면 abcde 를 잠시 다른 곳에 보관해놓았다가 입력이 끝난다면 이를 한꺼번에 처리하는 것입니다.

사실, 이 두 방법이 어떤 차이가 있을 수 있느냐 라고 물을 수 있지만 아래 비유를 보면 쉽게 이해가 될 것입니다. 우리가 만일 약수터에 가서 물을 떠온다고 해봅시다. 물을 3 L 받아 온다고 했을 때 우리는 물을 두 가지 방법으로 받아올 수 있습니다. 하나는 손에 물을 받아서 약수터까지 수십번 왔다 갔다 하는 것이고, 다른 방법은 양동이를 들고가서 3 L를 채운 후, 다시 양동이를 가지고 내려오는 것입니다.

자. 그럼 어떤 방법이 합리적인것 같습니까? 손으로 조금씩 조금씩 받아서 수십번 왔다갔다 하는 것이 나을 까요? 아니면 양동이를 들고 한 번만 갔다오는 것이 나을까요? 당연히, 후자가 훨씬 좋은 방법이겠지요. 컴퓨터도, 원리는 조금 더 복잡하지만 이러한 방법을 채택하고 있는 것입니다.

그렇다면 컴퓨터의 양동이에 해당하는 부분은 무엇일까요? 바로, **버퍼(buffer)** 라고 부르는 것입니다. 또한, 수 많은 버퍼 중에서도 키보드의 입력을 처리하는 버퍼는 바로 입력 버퍼, 혹은 stdin (흔히 입력 스트림) 이라 부르는 것입니다.

다시말해 우리가 키보드로 쳐다는 모든 정보는 일시적으로 stdin 에 저장되었다가 나중에 입력이 종료되면 한꺼번에 처리를 하는 것입니다. 그런데, 컴퓨터가 어떻게 우리가 입력을 종료했는지 알 수 있죠? 바로 엔터를 치면 됩니다. 왜냐하면 이전에도 우리가 계속 보았듯이 엔터를 치기만 하면 입력을 끝내고 프로그램이 계속 실행되었잖아요. 안그래요?

다시말해 컴퓨터는 개행 문자, 즉 \n 을 '입력을 종료하였으니 버퍼에 들어 있는 내용을 가지고 놀아라' 라는 뜻으로 받아 들입니다.[1] 그런데 컴퓨터는 \n 까지 버퍼에 저장하게 됩니다. 즉, 우리가 1 을 쓰고 엔터를 딱 치면 버퍼에 아래와 같은 상태가 됩니다.

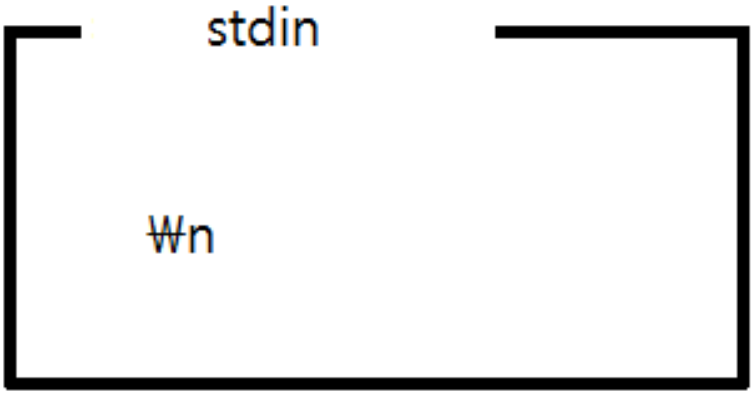
1) 잘 알고 게시고 있겠지만 \n 은 두 개의 문자 '\' 와 'n' 이 아니라, 하나의 문자 (개행문자 - 키보드 상의 엔터)를 의미합니다!

자. 그럼 입력을 끝냈다면 컴퓨터는 scanf 함수를 이용해서 stdin 으로 부터 숫자를 얻어옵니다. 왜 숫자냐면, 잘 아겠지만 우리가

```
scanf("%d", &num);
```

로 하였기 때문이죠. 즉 오직 숫자 데이터만 stdin 에서 얻어온다는 말입니다. 그렇다면 scanf 함수는 언제까지 stdin 으로 부터 데이터를 얻어올까요? 바로 ' ', '\n', '\t' 를 만날 때 까지 입니다. 여기서 ' ' 는 띄어쓰기 한 칸을 의미합니다. 또한 \t 는 여러분들이 아직 본적도 없고 제가 이야기 한 적은 없지만 TAB 문자로 여러분이 키보드에 TAB 키를 누르게 되면 이 문자를 쓰는 것입니다. 또한 \n 은 이야개 했던데로, 엔터 이죠. 다시 말해 scanf 함수는 stdin 에서 위 세 개의 문자들을 만난다면 '아. 여기서 입력은 끝이구나' 하고 입력을 종료해 버립니다.

참고적으로 %d 계열의 것들, 즉 수를 입력받는 형식은 수가 아닌 데이터가 와도 입력을 종료해 버립니다. 즉, a 를 입력했다면 num 에는 아무런 값이 들어가지 않아 치명적인 결과를 야기할 수 있습니다. 뿐만 아니라 수 데이터를 입력받는 형식의 경우 처음 부터 공백문자가 나타나면 수가 나타날 때 까지 입력을 계속 받게 됩니다. (다시 말해, 수를 입력 받는데 엔터를 아무리 쳐도 숫자를 치기 전까지 넘어가지 않는다) 암튼 scanf 함수는 공백 문자(' ', '\n', '\t') 를 만나기 전까지 stdin 에서 데이터를 가져간 후 버퍼에서 삭제해 버립니다. 다시말해, 위 scanf 함수가 num 에 1 을 저장한 후 버퍼의 모습은 아래와 같습니다.



자, 이제. 우리의 말을 아주 잘 듣는 컴퓨터는

```
scanf("%c", &c);
```

를 실행하게 됩니다. 그런데 말이죠, %c 는 이유를 불문하고 stdin 에서 딱 한개의 문자만을 가져오게 됩니다. 만일 stdin 에 아무것도 없다면 사용자의 입력을 기다리고 있겠지만 stdin 에 무언가가 있다면 그 것을 냉큼 가져오게 되지요. 그런데 공교롭게도 위에서 \n 을 버퍼에 남겨놓았기 때문에 scanf 는 냉큼 이를 c 에 저장하게 됩니다. 즉, c 에는 사용자의 입력을 받지도 않고 \n 을 집어 넣은 것이지요.

따라서 만일 우리가 printf("%c 출력", c); 를 해보게 된다면 '출력'이 한 칸 개행(엔터가 쳐져서)되어 나타나게 됩니다.

%s 로 scanf 에서 받을 경우

```c
/* 그렇다면 %s 는 ? */
#include <stdio.h>
int main() {
  char str[30];
  int i;

  scanf("%d", &i);
  scanf("%s", str);

  printf("str : %s", str);

  return 0;
}
```

성공적으로 컴파일 한다면

실행 결과

```
1
asdfasfasdf
str : asdfasfasdf
```

오오.. 이번에는 다행입니다. %c 와는 달리 %s 의 경우 컴퓨터가 사용자로 부터 입력을 잘 받았습니다. 사실, 그 이유는 간단합니다. 일단,

```
scanf("%d", &i);
```

를 실행하여 사용자로 부터 수를 입력 받게 된다면 역시 stdin 에는 \n 이 남아있게 됩니다. 그리고

```
scanf("%s", str);
```

를 실행하게 되면 역시 수 데이터를 입력 받는 형식 처럼실질 적인 데이터 (공백 문자가 아닌 것들)이 나오기 전 까지 버퍼에 남아 있던 공백 문자들은 무시하고 실질 적인 문자(공백 문자가 아닌 것들)가 입력이 된다면 그 다음 부터 등장하게 되는 공백 문자에서는 종료하게 됩니다. 즉, 기존에 1 을 입력하였을 때 남아있었던 \n 은 사라지고, 내가 aasdfdasfads 를 입력하고 난 후, 엔터를 쳤을 때 들어가는 \n 을 인식 하게 된다는 것이지요.

아무튼, 결론적으로 요약하자면 %s 나 %d 그리고 다른 모든 수 데이터를 입력 받는 형식은 버퍼에 신경 쓸 필요 없이 자유롭게 사용할 수 있습니다. 하지만 %c 를 이용할 때 에는 버퍼에 무엇이 남아 있는지 잘 고려해야 합니다. 이는 정말로 번거로운 일이 아닐 수 없습니다. 물론, 이를 대체할 수 있는 멋진 대안이 있지만 이는 조금 있다가 고려하도록 하고 다음 예제를 살펴 보도록 합시다.

```c
/* 마지막 stdin 예제 */
#include <stdio.h>
int main() {
  char str1[10], str2[10];

  printf("문자열을 입력하세요 : ");
  scanf("%s", str1);
  printf("입력한 문자열 : %s \n", str1);

  printf("문자열을 입력하세요 : ");
  scanf("%s", str2);
  printf("입력한 문자열 : %s \n", str2);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
문자열을 입력하세요 : hello
입력한 문자열 : hello
문자열을 입력하세요 : baby
입력한 문자열 : baby
```

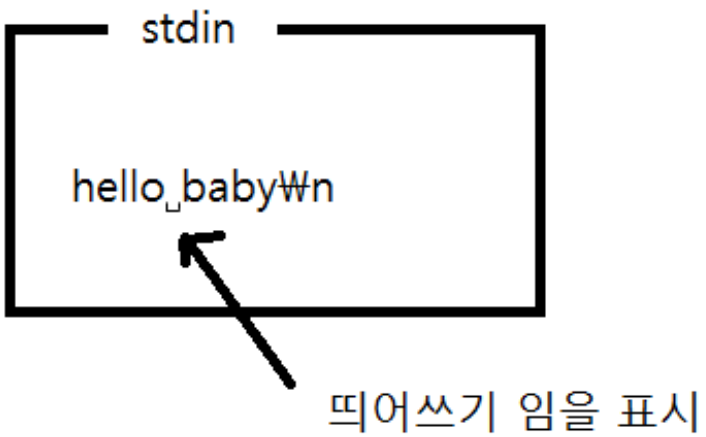
상당히 평범한 내용입니다. 여태까지의 강좌를 잘 따라오고 있었다면 위 내용쯤이야 쉽게 이해할 수 있을 것입니다.

그렇다면 다음과 같이 입력 해보도록 하겠습니다.

실행 결과

```
문자열을 입력하세요 : hello baby
입력한 문자열 : hello
문자열을 입력하세요 : 입력한 문자열 : baby
```

헉.. 이번에는 우리의 두번째 `scanf` 를 완전히 무시하고 지나갔습니다. 하지만 똑똑한 여러분이라면 왜 두번째 `scanf` 에서 사용자로 부터 입력을 받지 않았고, `str1` 에는 `hello`, `str2` 에는 `baby` 가 제대로 들어갔는지도 알 수 있을 것입니다. 우리가 "hello baby" 를 입력하였을 때 `stdin` 의 상태를 살펴봅시다.

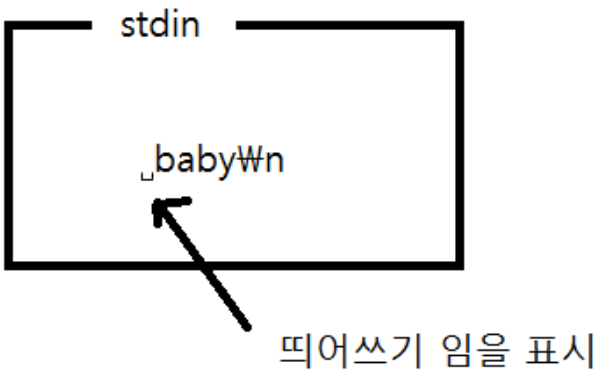


그렇다면

```
scanf("%s", str1);
```

scanf 함수는 stdin 으로 부터 **의미가 있는 문자 (공백 문자(' ', '\n', '\t') 를 제외한 나머지 문자)** 가 나올 때 까지 모든 공백 문자들을 무시합니다. 위의 경우 stdin 에서 처음에 공백 문자가 하나도 없으므로 바로 stdin 으로 부터 데이터를 가져오겠군요. 데이터를 가져오다가 공백 문자를 만나게 되면 입력을 중지합니다. 위의 경우 ' ' 이 공백 문자의 역할을 하기 때문에 str1 에는 hello 까지만 입력이 됩니다.

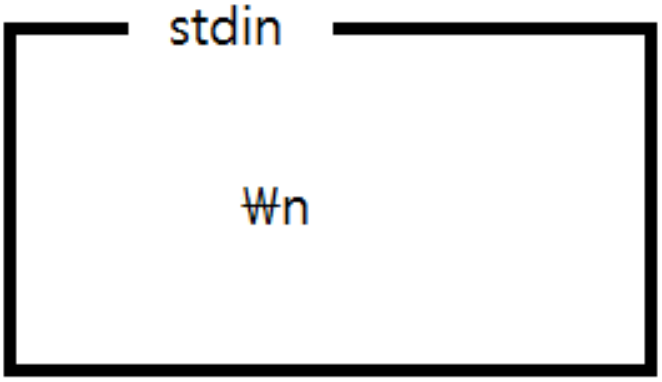
첫번째 scanf 함수를 지나게 되면 stdin 의 모습은 아래와 같습니다.



이제 두번째 scanf 를 지나갈 차례입니다.

```
scanf("%s", str2);
```

scanf 함수는 stdin 에 아무 것도 없거나, 공백 문자들 밖에 없다면 사용자가 무언가 의미 있는 문자를 입력해줄 때 까지 기다리겠지만 위 경우는 상황이 다릅니다. 일단, 처음에 공백 문자인 띄어쓰기는 살포시 무시합니다. 왜냐하면 아직 의미 있는 문자를 받지 않았기 때문이죠. 그 다음에 b 를 보고 str2 에 입력을 쭉 받기 시작합니다. 그러다가 마지막에 공백 문자인 \n 을 보고 입력을 중지합니다. 따라서 메모리에는 다음과 같이 \n 만이 덩그러니 남아있게 됩니다.



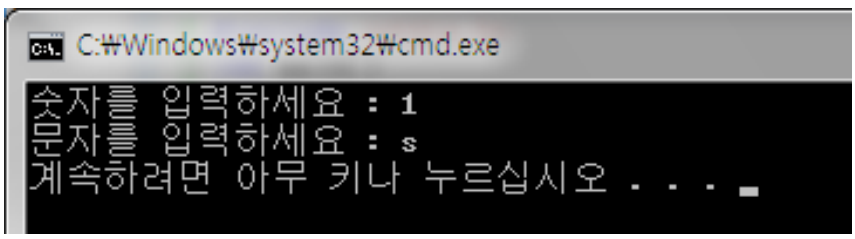
아무튼. scanf 는 상당히 이해하기 복잡한 것임은 틀림이 없습니다. 가뜩이나 머리 아픈데 %c 를 이용하면 고려해야 될 것이 더욱 많아지니 정말 짜증이 나는 것 같습니다. 하지만 다행스럽게도 이러한 문제를 해결할 수 있는 방법이 있을 뿐더러 실질 적으로 %c 는 많이 쓰이지 않으니 다행인 것 같습니다.

도대체 이 문제를 어떻게 해결하냐

하지만, 아무리 %c 를 사용하지 않는다고 해도 필연적으로 사용할 일이 생기게 됩니다. 그렇다면 그 때 마다 이처럼 버퍼에 \n 이 남아 있는 것을 고려해야 할까요? 정말 번거로운 일이 아닐 수 없습니다. 하지만 걱정 마십시오. 이를 해결할 수 있는 방법이 여러 가지가 있습니다.

```c
/*

버퍼 비우기

주의하실 점은 반드시 MS 계열의 컴파일러로 컴파일 해주세요. 즉, Visual Studio
계열의 컴파일러로 말이죠. 이 말이 무슨 말인지 모르면 그냥 늘 하던대로 하면
됩니다.

gcc 에서는 정상적으로 작동되지 않는 위험한 코드 입니다.

*/
#include <stdio.h>
int main() {
  int num;
  char c;

  printf("숫자를 입력하세요 : ");
  scanf("%d", &num);

  fflush(stdin);

  printf("문자를 입력하세요 : ");
  scanf("%c", &c);
  return 0;
}
```

성공적으로 컴파일 했다면



아마도 여러분은 컴파일 하면서 포스 넘치는 주석을 보며 무언가 당황 하셨을 수 도 있습니다. 하지만 걱정하진 마세요. 지금 수준의 프로그래밍에서는 크게 걱정할 문제는 아닙니다. 먼저 위 소스가 어떻게 해서 올바르게 작동하는지 부터 살펴보도록 합시다. 사실, 올바르게 라는 말 보다는 'scanf 가 사용자의 입력을 무시하지 않는지' 가 적당할 듯 하네요.

```
printf("숫자를 입력하세요 : ");
scanf("%d", &num);
```

위까지 실행했을 때 에는 이전처럼 stdin 에 '\n' 이 남아 있습니다. 그런데 말이죠.

```
fflush(stdin);
```

두둥. 새로운 문장이 등장했습니다. 위 문장의 의미는 'stdin 을 비워버려라' 라는 의미이죠. 다시 말해 stdin 에 있는 모든 데이터들을 날려버리게 되는 것입니다. 따라서 버퍼가 완전히 비워지게 됩니다. 즉 버퍼에 가시 처럼 남아 있던 \n 이 사라지게 됩니다.

```
scanf("%c", &c);
```

따라서 그러한 상태에서 scanf 를 호출하게 되면 %c 는 버퍼에 아무것도 남아 있는 것이 없으므로 사용자의 입력을 차분히 기다리고 있게 됩니다. 즉, 우리가 c 에 원하는 값을 넣을 수 있다는 뜻이 되죠. 하지만 프로그램 코드 상단에 있는 무서운 주석을 보면 알겠지만 사실 위 코드는 추천하고 싶지 않습니다. 왜냐하면 fflush 가 표준으로 '무슨 역할은 한다' 라고 정해진 것이 아니기 때문입니다.

다시 말해 우리의 Visual Studio 에선 fflush 함수가 버퍼를 비우는 훌륭한 역할을 하지만 다른 것 - 예를 들면 gcc 같은 데에서는 이러한 작업을 하지 않을 가능성이 매우 매우 큽니다. 다시 말해, 위 방법은 그다지 권장하고 싶은 방법은 아니지만 적어도 우리의 수준에서는 정확하게 작동하고 편리하기 때문에 많이 사용합니다.[2)]

```
/* getchar 함수 이용 */
#include <stdio.h>
int main() {
  int num;
  char c;

  printf("숫자를 입력하세요 : ");
  scanf("%d", &num);

  getchar();

  printf("문자를 입력하세요 : ");
```

2) Visual Studio 2015 부터는 표준을 따라서 fflush 가 위와 같이 작동하지 않습니다.

```
  scanf("%c", &c);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
숫자를 입력하세요 : 1
문자를 입력하세요 : c
```

오. 이번에도 제대로 작동하고 있습니다.

```
printf("숫자를 입력하세요 : ");
scanf("%d", &num);
```

일단, 위 부분까지만 실행하면 역시 stdin 에는 \n 이 남아있게 됩니다. 상당히 곤란한 일이죠.

```
getchar();
```

그렇데 말이죠, 우리가 getchar 이라는 함수를 호출했습니다. 이 함수의 역할은 'stdin 에서 한 문자를 읽어와서 그 값을 리턴한다' 입니다. 물론 한 문자를 읽어오면 읽어온 문자는 stdin 에서 사라지게 되지요. 따라서 위 함수를 호출 함으로써 \n 을 stdin 에서 읽어와 지워버릴 수 있는 것이지요.

만일 우리가

```
ch = getchar();
prinf("%c", ch);
```

을 해서 getchar 함수가 리턴한 값을 출력해보았다면 화면상에 한 칸 엔터(== \n)가 쳐진 것이 출력될 것입니다. (여러분이 한 번 해보세요~) 이제, 버퍼가 비워진 상태에서 scanf 함수를 호출하게 되면 성공적으로 사용자의 입력을 받게 되는 것입니다. 상당히 단순하지요?

getchar 함수를 호출한 방법은 여러 모로 많이 쓰이는 방법 입니다. 기본적으로 scanf 에서 %c 형식을 사용하는 것을 권하고 싶지는 않지만 정 사용하고자 한다면 getchar() 을 scanf 이전에 호출해서 버퍼를 비워주기 바랍니다. 그런데 말이지요. 위 방법도 문제가 어지간히 있습니다. 만일 버퍼에 한 문자만 남겨져 있는 것이 아니면 어떡할까요? 한 번 숫자를 입력할 때 123abc 를 쳐 보았습니다.

```c
/* c 에 무엇이 들어가는지 살짝 보아야 하므로 코드를 약간 수정했습니다 */
#include <stdio.h>
int main() {
  int num, i;
  char c;

  printf("숫자를 입력하세요 : ");
  scanf("%d", &num);

  getchar();

  printf("문자를 입력하세요 : ");
  scanf("%c", &c);

  printf("입력한 문자 : %c", c);
  return 0;
}
```

성공적으로 컴파일 했다면

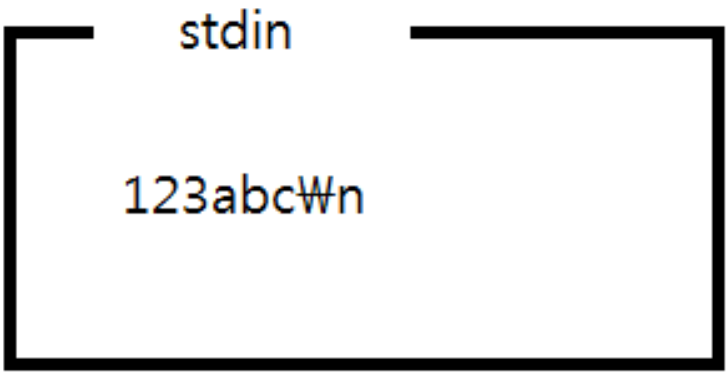
실행 결과

```
숫자를 입력하세요 : 123abc
문자를 입력하세요 : 입력한 문자 : b
```

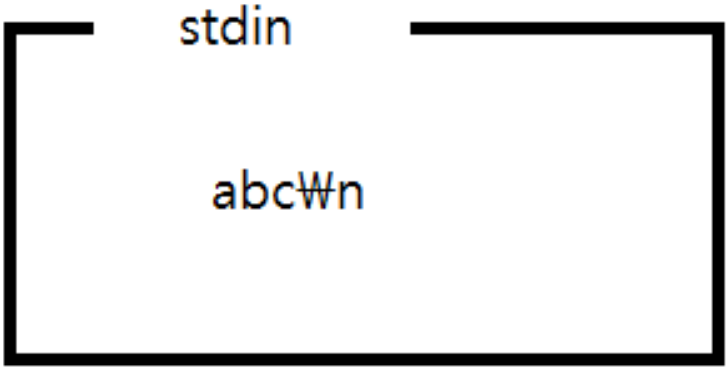
아.. 역시 제가 우려했던 대로 scanf 에서 사용자의 입력을 기다리지 않고 지나쳐 버렸습니다. 뿐만 아니라 c 에도 우리가 원하지 않던 b 라는 값이 들어가 있습니다. 도대체 왜 이런 일이 발생한 것일까요? 일단 버퍼에서 무슨 일이 벌어지고 있는지 차근 차근 살펴보도록 합시다.

```c
printf("숫자를 입력하세요 : ");
scanf("%d", &num);
```

일단 위 코드가 실행되어서 사용자로 부터 입력을 기다립니다. 사악한 Psi 는 123abc 를 쳤습니다. 그렇다면 버퍼에 다음과 같이 들어가겠지요.



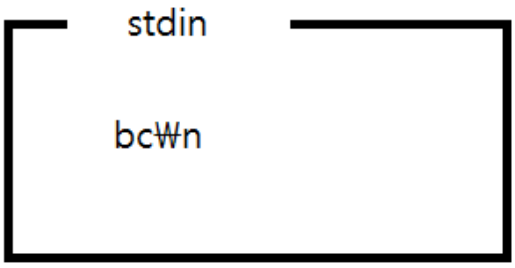
이제, scanf 함수가 stdin 에서 차례 차례 데이타를 읽어옵니다. 그 때도 말했들이 데이타를 읽어올 때 공백문자나 숫자가 아닌 것들을 만나게 되면 stdin 에서 부터 그만 읽어온다고 했죠? 이 때 a 가 숫자가 아니기 때문에 123 까지 읽은 후 stdin 에서 부터 그만 읽어 옵니다. 따라서 stdin 은 다음과 같은 모습이 되겠군요.



그렇다면 아래의 문장이 실행됩니다.

```
getchar();
```

이는 이전의 문제점을 말끔히 해결해 주었죠. stdin 으로 부터 한 문자를 얻어오는 방법으로 말이지요. 여기서도 getchar 은 똑같은 역할을 수행합니다. 즉 stdin 으로 부터 한 문자, 위 경우 a 를 읽어옵니다.



아 이런. 버퍼가 깔끔하게 비워지지 않았습니다. 이러한 우려 속에서 아래의 코드가 실행됩니다.

```
printf("문자를 입력하세요 : ");
scanf("%c", &c);
```

음.. scanf 의 입장에서 버퍼에 읽어올 것들이 잔뜩 있으니 행복할 것 같습니다. 버퍼에서 한 문자를 읽어 옵니다. 그것이 바로 b 가 됩니다. 따라서 c 에는 우리가 원하지 않던 b 가 들어가게 됩니다. 그리고 물론 b 는 stdin 에서 사라지게 되죠. 다음에 또 scanf("%c", &c); 를 하게 되면 이번에는 c 가, 한 번 더하면 \n 이 읽어지겠죠?

아무튼. 여기서 내릴 수 있는 결론은 "되도록이면 %c 를 사용하지 말자" 입니다. scanf 에서 %c 를 사용하는 것은 정말로 권장하고 싶지 않은 일입니다. 만일 정말로 문자 하나만을 입력받는 프로그램을 만드려면 scanf 에서 %s 형태로 문자열을 입력 받은 뒤에 맨 앞의 한 문자만 취하는 식으로 만들면 되겠습니다.

결론 : 문자 대신 문자열을 입력 받도록 하자!

생각해보기

문제 1

키보드로 부터 입력을 받는 함수는 scanf 나 getchar 말고도 여러가지가 있습니다. 이들에 대해 조사해 보는 것이 어떨까요? (난이도 : 無 - 쉬운 것도 아니고 어려운 것도 아님)

문제 2

화면에 출력하는 함수도 `printf` 만 있는 것이 아닙니다. 화면에 출력하는 함수에 대해서 알아보는 것이 어떨까요?(난이도 : 無)

문자열 리터럴에 대한 이해

안녕하세요~ 여러분. 문자열의 3 번째 강의 입니다. 아마도 지난번 강의에서 좌절을 느끼신 분들은 아마 이번 강의는 아주 아주 수월하게 이해해 나갈 수 있으리라 믿고 있습니다. 끝이 보이지 않는 C 언어 강좌 이지만 이제 거의 70 ~ 80% 정도를 지나 왔다고 해도 무방 합니다. 문자열이 끝나게 되면 구조체에 대해 다루게 되는데, 구조체가 끝나면 잡다한 것들만 남아서 슝슝슝 지나갈 수 있습니다. 또한 조금만 더 지나면 C 언어에 대한 강좌 보다는 여러가지 프로그램을 만들어 보며 C 언어를 익혀 보는 강좌를 중심으로 진행할 것입니다.

인터넷 강좌 치고 체계적인 것 같죠? 아무튼. 15 - 3 강, 전체 강좌 수로 치면 25 번째 강좌를 시작하겠습니다.

일단, 아래 코드를 실행해봅시다.

```c
/* 문자열 */
#include <stdio.h>
int main() {
  char str[] = "sentence";
  char *pstr = "sentence";

  printf("str : %s \n", str);
  printf("pstr : %s \n", pstr);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
str : sentence
pstr : sentence
```

와 같이 나옵니다.

```c
char str[] = "sentence";
char *pstr = "sentence";
```

일단, 여러분들은 당연하게도 위 두 개의 문장을 보고 이상하다고 생각하셨을 것입니다. 일단 첫번째 문장은 평범한 문장 입니다. `sentence` 라는 문자열을 `str` 이라는 배열에 집어 넣고 있지요. 그런데 두 번째 문장은 말이죠. 상당히 이상합니다. 왜냐하면 일단 *"sentence"* 는 문자열이고, 어떤

변수의 주소값이 아닙니다. pstr 는 char 형을 가리키는 포인터 이므로 char 형 변수의 주소값이 들어가야되기 때문이죠.

그런데 우리는 마치 *"sentence"* 를 특정한 주소값 마냥 사용하고 있습니다. 그런데, 말이죠. *"sentence"* 는 주소값 맞습니다. 그렇다면 무엇의 주소값이죠? 바로, *"sentence"* 라는 문자열이 저장된 주소값 (시작 주소값) 을 말합니다. 정말로 놀랍지 않습니까? 사실 저도 잘 믿기지 않습니다. 만일 믿기지 않는다면 아래 문장을 넣어 실행해 보세요.

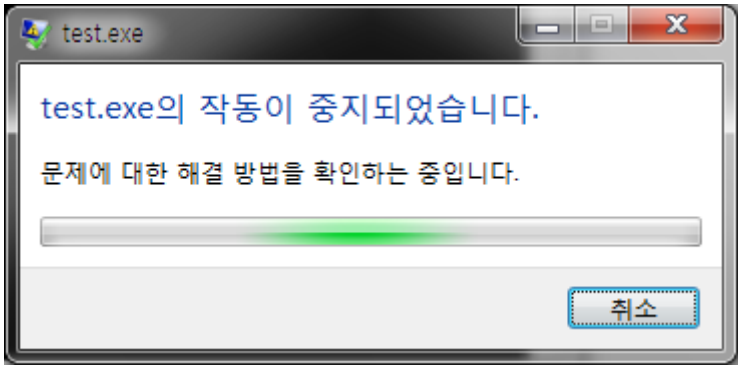
```
printf("%d \n", "sentence");
```

정말로, 특정한 수가 출력됨을 알 수 있습니다. 그렇다면 이 *"sentence"* 는 도대체 뭘까요?

일단, *"sentence"* 의 정체를 먼저 파악하기 전에 다음의 소스 코드를 실행해보시기 바랍니다.

```
/* 문자열 */
#include <stdio.h>
int main() {
  char str[] = "hello";
  char *pstr = "goodbye";

  str[1] = 'a';
  pstr[1] = 'a';

  return 0;
}
```

성공적으로 컴파일 했어도 실행해 보면 오류가 날 것입니다.



Windows 7 의 경우 위와 같은 화면이 나오며 다른 운영체제의 경우 다른 화면이 나올 수 있습니다.

헐.. 왜 오류가 난 것일까요? 일단, `pstr[1] = 'a';` 를 주석 처리한 후 다시 실행해 보면 제대로 실행됨을 알 수 있습니다. 다시말해,

```
pstr[1] = 'a';
```

가 문제인 것이군요. 그런데 말이죠. 왜 문제가 발생한 것일까요? 맨 위의 예제에서 `pstr` 의 값을 읽기만 하였을 때(`printf` 함수는 값을 읽기만 하지 변경하지 않는다) 정상적으로 실행되었지만 아래에서 `pstr[1] = 'a';` 를 통해 `pstr` 의 값을 변경하였을 때 오류가 출력된 것을 보아 마치 상수 처럼 컴퓨터에서 값을 변경하도록 허락 하지 않는 것 같습니다.

리터럴(literal)

프로그래밍 언어에서 **리터럴(literal)**이란, 소스 코드 상에서 고정된 값을 가지는 것을 일컫습니다. 특히, C 언어의 경우 큰 따옴표(") 로 묶인 것들을 **문자열 리터럴(string literal)** 이라 부릅니다.

```
char *pstr = "goodbye";
printf("why so serious?");
scanf("%c", str[0]);
```

그렇다면 위 3 개의 문장에서 문자열 리터럴은 무엇일까요? 물론, 짐작하였던 대로 `goodbye`, `why so serious`, `%c` 모두 리터럴이 됩니다.

컴퓨터는 이러한 리터럴들을 따로 모아서 보관합니다. 즉, 프로그램을 실행하면 메모리 상의 특별한 곳에 `goodbye`, `why so serious`, `%c` 와 같은 리터럴들이 쭈르륵 보관되어 있는 공간이 생긴다는 것입니다.[3)]

따라서 `char *pstr = "goodbye";` 을 실행하게 되면 컴퓨터는 *"goodbye* 의 시작 주소값 좀 가져와서 *pstr* 에 대입좀 해라*"* 라는 의미의 작업을 실행합니다. 따라서, `pstr` 은 *"goodbye"* 라는 리터럴을 가리키고 `printf("%s", pstr)` 을 했을 때 `goodbye` 를 성공적으로 출력할 수 있게 되었던 것이죠.

그런데 말이죠. 아까 위에서 이야기 하였던 리터럴의 조건 기억 하시나요? 아까 분명히 리터럴은 소스 코드 상에서 고정된 값을 가지는 것들 이라고 이야기 했습니다. 다시 말해서 실제 프로그램 실행 중에서도 리터럴의 값은 절대로 변경 되서는 안된다는 것입니다.

아까도 말했듯이 만일 `hello` 라는 리터럴의 값을 실수로 (물론 내가 했을 수도 있고 컴퓨터의 버그로 그랬을 수 도 있고) `hi` 로 변경하였다면 사용자는 분명히 `str` 에 `hello` 라는 값을 넣으라고 명령했지만 `hi` 가 들어가게 되어 큰 문제를 야기할 수 있게 됩니다.

따라서 리터럴이 보관되는 곳은 **오직 읽기만 가능한 곳** 이 됩니다. 만일 이곳을 함부로 변경하려고 하는 시도가 있다면 바로 프로그램이 강제로 종료되게 됩니다.[4)]

3) 좀 더 궁금하신 분들을 위해 추가적으로 설명하자면, 프로그램이 실행되서 메모리에 로드되면, 5 가지 종류의 영역 (text segment, data segment, bss segment, heap, stack) 이 존재합니다. 이 때, 텍스트 세그먼트(text segment) 에 프로그램 코드와 상수, 리터럴 등이 여기서 정의됩니다. 왜냐하면 텍스트 세그먼트에 있는 내용들은 읽기만 가능하기 때문이지요. 물론 이 사실은 컴파일러 구현에 따라, 사용하는 운영체제 환경에 따라서 다를 수 있습니다.

4) 이 사실 역시 사용하는 운영체제, 환경에 따라 다릅니다.

그렇기 때문에 우리는 `char str[] = "hello";` 를 했다면 `str` 에 `hello` 가 들어가고 `printf ("why so serious?");` 를 했다면 화면에 `why so serious` 가 출력될 것이라고 보장할 수 있다는 것이죠. 왜냐하면 이 모든 문자열들이 "문자열 리터럴" 이라는 이름 하에 메모리 상의 특별한 공간에서 보호 받고 있기 때문입니다.

```
char *pstr = "goodbye";
pstr[1] = 'a';
```

그럼 위 코드를 다시 살펴봅시다. 우리는 앞서 `goodbye` 역시 문자열 리터럴 이기 때문에 "리터럴들의 세상 (정확히 말하면 이 곳에는 리터럴들만 있는 것이 아니라 우리가 프로그램 상에서 정의한 상수들도 이곳에 저장됩니다) " 에 저장된다고 했습니다. 그런데 이 곳은 오직 읽기만 가능한 곳이므로 쓰기를 시도 하려고 할 시에 오류를 뿜게 된다고 했습니다. 그런데 말이죠. 제가 무례하게도 `pstr[1] = 'a';` 을 통해 *"리터럴 세상에 저장된 리터럴 goodbye"* 의 값을 변경하려고 했습니다. 따라서 컴퓨터에서 오류를 뻑 뿜게 되는 것이죠.

반면에

```
printf("pstr : %s \n", pstr);
```

와 같이 오직 읽기 작업만을 수행하는 `printf` 의 경우 잘 실행되지요. 왜냐하면 리터럴 세상에 저장된 리터럴들에 쓰기는 불가능 하지만 적어도 읽기는 허용되기 때문입니다.

그렇다면 아래 코드는 어떨까요?

```
char str[] = "hello";
```

사실은 위 *"hello"* 는 리터럴이라고 부르기 애매합니다. 왜냐하면 위 배열의 정의는 사실 컴파일러에서 아래와 같이 해석되기 때문이지요.

```
char str[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

이는 그냥 `str` 이라는 배열에 hello 라는 문자열을 복사하게 될 뿐입니다. 그리고 위 배열은 텍스트 세그먼트가 아니라 스택(stack)이라는 메모리 수정이 가능한 영역에 정의가 됩니다. 따라서 `str` 안의 문자열은 수정이 가능합니다.

주의 사항

참고적으로 VS 2017 이상에서는 리터럴을 `char*` 가 가리킬 수 없습니다. 반드시 `const char*` 가 가리켜야 하며, 덕분에 리터럴을 수정하는 괴랄한 짓을 컴파일 단에서 막을 수 있습니다.

문자열 다시 가지고 놀기

C 언어에서 문자열을 다루는 일은 생각보다 불편한 편입니다. 예를 들어서 `int` 형 변수의 경우

```
int i, j = 0;
i = j + 3;
```

과 같이 값을 더하는 것이 가능하지만 문자열의 경우

```
char str1[] = {"abc"};
char str2[] = {"def"};
str1 = str1 + str2;
```

를 한다고 해서 `str1` 이 "abcdef" 가 되는 것이 절대로 아니지요. `str1 + str2` 는 각 배열의 주소값을 더하는 것인데, 이전에도 말했듯이 배열의 이름은 포인터 상수 이기 때문에 대입 연산을 수행시 오류가 나게 됩니다.

뿐만 아니라 다음과 같이 문자열을 비교하는 것도 불가능합니다.

```
if (str1 == str2) ```

왜냐하면 위 문장의 의미는 "`str1` 의 문자열이 들어있는 메모리 상의 (시작)주소와 `str2` 의 문자열이
↪ 들어있는 메모리 상의 (시작) 주소값을 비교해라" 라는 의미의 문장이기 때문입니다. 따라서 역시 우리가
↪ 원하던 기능이 실행 될 수가 없습니다. 물론 다음과 같은 문장도 원하는 대로 실행이 되지 않습니다.
```

if (str1 == "abc")

잘 알겠지만 "abc" 은 리터럴 입니다. 즉, `str1` 과 "abc" 를 비교한다는 뜻은 "str1 이 저장된 메모리 상의 주소값과 `abc` 라는 문자열 리터럴이 보관된 메모리 상의 주소값을 비교" 하는 문장이기 때문에 절대로 우리가 원하는 "str1 의 문자열과 `abc` 를 비교한다" 라는 뜻을 가질 수 없습니다.

가장 짜증나는 문제는 문자열을 원하는 대로도 복사를 못한다는 것입니다. 다시말해 `int` 형 변수처럼 원하는 값을 "대입" 할 수 없다는 말입니다. 만일 우리가

```
str1 = str2;
```

라는 문장을 쓴다라면 "`str1` 에 `str2` 의 값을 대입해라" 라는 문장이 되는데 역시 `str1` 의 값은 바뀔 수 없는 포인터 상수 이기 때문에 오류가 발생하게 됩니다. 여하튼 문자열을 다루는데에는 제약이 너무나 많습니다. 하지만 다행스럽게 함수를 이용해서 그나마 편리하게 다룰 수 있습니다.

일단, 위에서 지적한 내용을 바탕으로 문자열을 자유롭게 다루려면 다음과 같은 함수들이 필요할 것입니다.

- 문자열 내의 총 문자의 수를 세는 함수
- 문자열을 복사하는 함수
- 문자열을 합치는 함수 (즉 더하는)
- 문자열을 비교하는 함수

제 강좌에서는 위 4 개의 함수들을 모두 구현해 보도록 할 것입니다. (1 번의 경우 15 -1강에서 한 내용이므로 생략하도록 하겠습니다) 제가 이를 모두 구현하기 전에 여러분들이 한 번 어떻게 하면 만들 수 있는지 생각해 보도록 했으면 합니다.

문자열을 복사하는 함수

문자열을 복사하는 함수는 어떻게 하면 만들 수 있을까요? 우리가 무언가를 작업하는 함수를 만들기 전에 반드시 고려해야 하는 사항들은 다음과 같습니다. (이 사실을 이전 함수 단원에서 이야기 했으면 더 좋았을 것을..)

1. 이 함수는 무슨 작업을 하는가? (자세할 수록 좋다)
1. 함수의 리턴형이 무엇이면 좋을까?
1. 함수의 인자으로는 무엇을 받아야 하는가?

특히 ① 번의 경우 상당히 중요합니다. "무슨 무슨 함수를 만들어야 겠다" 라고 정하지도 않고 무턱대고 함수를 만들다 보면 소스 코드가 상당히 난잡해지고 이해하기 힘들게 됩니다. 이 경우 우리는 말그대로 문자열을 복사하는 함수, 즉 a 라는 문자열이 있다면 a 문자열의 모든 내용을 b 로 복사하는 함수 입니다.

두번째로 함수의 리턴형을 생각해봅시다. 문자열을 복사하는 함수에서 무슨 리턴형이 필요하냐고 물을 수도 있는데 저의 경우 복사가 성공적으로 되었다면 1 을 리턴하도록 만들어보고 싶습니다. 즉 `int` 형의 함수를 만들 것 입니다.

세번째로 함수의 인자로 무엇을 받아야 할 지 생각해 봅시다. 당연하게도 두 개의 문자열을 받아야 하므로 포인터를 사용해야겠죠? 이 때 문자열들은 `char` 형 배열 이기에 `char*` 을 인자로 2 개 가지는 함수를 만들 것 입니다.

```
/*

int copy_str(char *dest, char *src);

src 의 문자열을 dest 로 복사한다. 단, dest 의 크기가 반드시 src 보다 커야 한다.
```

```
*/
int copy_str(char *dest, char *src) {
  while (*src) {
    *dest = *src;
    src++;  // 그 다음 문자를 가리킨다.
    dest++;
  }
  *dest = '\0';

  return 1;
}
```

예를 들어 위 함수를 써먹어 봅시다.

```
/* copy_str 사용 예제 */
#include <stdio.h>
int copy_str(char *src, char *dest);
int main() {
  char str1[] = "hello";
  char str2[] = "hi";

  printf("복사 이전 : %s \n", str1);

  copy_str(str1, str2);

  printf("복사 이후 : %s \n ", str1);

  return 0;
}
int copy_str(char *dest, char *src) {
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }

  *dest = '\0';

  return 1;
}
```

성공적으로 컴파일 했다면

실행 결과

```
복사 이전 : hello
복사 이후 : hi
```

현재 여러분 정도의 수준이 되었다면 위 copy_str 함수 정도는 손쉽게 분석할 수 있으리라 믿지만 그래도 만약을 위해서 한 번 설명 해보도록 하겠습니다.

```
while (*src) {
  *dest = *src;
  src++;
  dest++;
}
```

먼저 while 문 부분을 살펴봅시다. while 문의 조건이 *src 입니다. 뭔 뜻인지 알겠죠? 문자열을 다룰 때 많이 쓰는 방법인데, NULL 문자의 값이 0 이므로 *src 가 NULL 문자에 도달하기 전 까지 while 문이 계속 돌아가게 됩니다.

그리고 *dest = *src 를 통해서 src 의 문자를 dest 에 대입하였습니다. 그리고 src 와 dest 를 각각 1 씩 증가시켰는데.. 포인터의 연산 기억 하시죠? 포인터에 1 을 더하면 단순히 주소값이 1 이 들어가는 것이 아니라 포인터가 가리키는 타입의 크기를 곱한 만큼 증가한다는 사실. 다시말해 배열의 그 다음 원소를 가리킬 수 있다는 것입니다.

```
*dest = '\0';
```

마지막으로 dest 에 '\0' , 즉 NULL 문자를 집어 넣었습니다. 아까 위의 while 문에서 src 가 NULL 이 된다면 while 문을 종료해 버렸기 때문에 src 에 넣을 틈이 없었는데 마지막에 위와 같이 처리해줌으로써 dest 에 NULL 문자를 끝부분에 삽입할 수 있게되었습니다.

참고적으로 이야기 하지만 이 함수는 상당히 위험한 편인데 왜냐하면 dest 의 크기가 src 의 크기보다 큰지 작은지 검사하지 않기 때문입니다. 만일 dest 의 크기가 src 보다 작다면 메모리의 허락 되지 않는 공간까지 침범하므로 큰 문제를 야기할 수 있습니다.

잠깐만요! 아마도 이 문자열을 복사하는 함수를 만들면서 "굳이 이 함수를 만들어야 되나?" 라고 생각하시는 분들이 있나요? 아마 있겠지요. 저도 그랬으니까요. 보통 이런 생각을 하시는 분들은 다음과 같은 코드를 제안합니다.

```
char str[100];
str = "abcdefg"; /* str 에 abcdefg 가 복사되지 않을까? */
```

그러나 이 방법으로 컴파일을 하게 되면 아래와 같은 오류를 만나게 됩니다.

컴파일 오류

```
error C2106: '=' : 왼쪽 피연산자는 l-value이어야 합니다.
```

도대체 왜 그런 것일까요? 아마 리터럴과 배열을 제대로 이해한 사람이라면 쉽게 답을 알 수 있을 것입니다. 일단, `str = "abcdefg"` 라는 문장은 'str 에 문자열 리터럴 abcdefg 가 위치한 곳의 주소값을 넣어라' 입니다. 그런데 말이죠. 우리가 이전에 배열에 대해 공부한 바로는 배열 이름은 상수 입니다. 즉, 배열의 주소값을 바꿀 수 없다는 것입니다!

따라서, 위와 같은 코드는 상수에 값을 대입하는 의미이기 때문에 오류가 발생하게 됩니다.

그런데 말이죠. 왜 다음 문장은 말이 되는 것일까요?

```
char str[100] = "abcdefg";
```

이는 단순히 C 언어에서 사용자의 편의를 위해 제공하는 방법이라 생각하면 됩니다. 오직 배열을 정의할 때 사용할 수 있는 방법이죠. 기억하세요!오직 배열을 정의할 때 에만 위 방법을 사용할 수 있습니다. 위처럼 사용하면 우리가 예상하던 대로 `str` 의 각각의 원소에 a 부터 g 까지 들어가게 됩니다.

문자열을 합치는 함수

문자열을 합치는 함수라 하면 다음과 같은 작업을 하는 함수를 말합니다.

```
char str1[100] = "hello my name is ";
char str2[] = "Psi";

stradd(str1, str2);

// str1 은 "hello my name is Psi" 가 된다.
```

한 번 만들어보세요.

완성된 소스는 아래와 같습니다.

```
/*

stradd 함수

dest 에 src 문자열을 끝에 붙인다.
이 때 dest 문자열의 크기를 검사하지 않으므로 src 가 들어갈 수 있는 충분한 크기가
있어야 한다.

*/
int stradd(char *dest, char *src) {
  /* dest 의 끝 부분을 찾는다.*/
  while (*dest) {
    dest++;
  }
```

```
  /*
  while 문을 지나고 나면 dest 는 dest 문자열의 NULL 문자를 가리키고 있게 된다.
  이제 src 의 문자열들을 dest 의 NULL 문자 있는 곳 부터 복사해넣는다.
  */
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }

  /* 마지막으로 dest 에 NULL 추가 (왜냐하면 src 에서 NULL 이 추가 되지
   * 않았으므로) */
  *dest = '\0';

  return 1;
}
```

이제 위 함수를 써먹어 봅시다.

```
#include <stdio.h>
int stradd(char *dest, char *src);
int main() {
  char str1[100] = "hello my name is ";
  char str2[] = "Psi";

  printf("합치기 이전 : %s \n", str1);

  stradd(str1, str2);

  printf("합친 이후 : %s \n", str1);

  return 0;
}
int stradd(char *dest, char *src) {
  /* dest 의 끝 부분을 찾는다.*/
  while (*dest) {
    dest++;
  }

  /*
  while 문을 지나고 나면 dest 는 dest 문자열의 NULL 문자를 가리키고 있게 된다.
  이제 src 의 문자열들을 dest 의 NULL 문자 있는 곳 부터 복사해넣는다.
  */
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }
```

```
  /* 마지막으로 dest 에 NULL 추가 (왜냐하면 src 에서 NULL 이 추가 되지
   * 않았으므로) */
  *dest = '\0';

  return 1;
}
```

성공적으로 컴파일 했다면

실행 결과

```
합치기 이전 : hello my name is
합친 이후 : hello my name is Psi
```

역시. 제대로 출력이 됩니다. 일단 stradd 의 구조는 단순합니다. dest 의 끝에 문자열을 덧붙이기 위해서는 먼저 dest 문자열의 끝 부분을 찾아야겠죠? 따라서

```
while (*dest) {
  dest++;
}
```

를 통해서 dest 의 널문자의 위치를 찾습니다. (물론 그 위치는 dest 가 가리키고 있겠지요) 이제, 그 널문자가 들어갔던 곳을 포함하여 dest 의 끝에 src 문자열을 덧쓰면 됩니다. 아래와 같이죠.

```
while (*src) {
  *dest = *src;
  src++;
  dest++;
}
```

물론 이때도 주의해야 할 점은 *src 가 NULL 이 되면 while 문이 종료되므로 src 의 널문자를 복사할 수 없게 됩니다. 따라서 아래와 같이 dest 의 끝부분에 NULL 문자를 집어 넣어주어야 합니다.

```
*dest = '\0';
```

자. 그럼 이해가 되셨는지요?

문자열을 비교하는 함수

문자열을 비교하는 함수라 하면 다음과 같은 작업을 하는 함수를 의미합니다.

```
if (compare(str1, str2)) {
  /*
   만일 str1 과 str2 가 같다면 이 부분이 실행되고 아니면 지나갑니다.
  참고로 if 문에서 0 이 아닌 값만 들어가면 무조건 참으로 처리되는 사실은 알고
  계시죠?
  */
}
```

한 번 만들어보세요.

완성된 소스는 아래와 같습니다.

```
int compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
```

이제 위 함수를 써먹어 봅시다.

```
#include <stdio.h>
int compare(char *str1, char *str2);
int main() {
  char str[20] = "hello every1";
  char str2[20] = "hello everyone";
  char str3[20] = "hello every1 hi";
  char str4[20] = "hello every1";

  if (compare(str, str2)) {
    printf("%s 와 %s 는 같다 \n", str, str2);
  } else {
    printf("%s 와 %s 는 다르다 \n", str, str2);
  }

  if (compare(str, str3)) {
    printf("%s 와 %s 는 같다 \n", str, str3);
  } else {
    printf("%s 와 %s 는 다르다 \n", str, str3);
```

```
  }

  if (compare(str, str4)) {
    printf("%s 와 %s 는 같다 \n", str, str4);
  } else {
    printf("%s 와 %s 는 다르다 \n", str, str4);
  }

  return 0;
}
int compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
hello every1 와 hello everyone 는 다르다
hello every1 와 hello every1 hi 는 다르다
hello every1 와 hello every1 는 같다
```

compare 함수가 어떻게 작동하는지 알아보도록 합시다,.

```
while (*str1) {
  if (*str1 != *str2) {
    return 0;
  }

  str1++;
  str2++;
}
```

일단 while 문에서 str1 의 끝에 도달할 때 까지 각 문자들을 비교합니다. 만일 한 문자라도 다르다면 if 문에 의해 0 이 리턴되고 함수는 종료됩니다. 그렇지 않다면 while 문을 끝가지 통과하게

되죠.

그런데 여기서 끝난 것이 아닙니다. 만일 `str1` 과 `str2` 가 `str1` 부분만 일치하였다면 어떨까요? 다시말해 `str1` 은 "abc" 이지만 `str2` 는 "abcd" 라면? 그렇다면 `while` 문에서 검사할 때 `str1` 이 끝날 때 까지만 검사하므로 `while` 문을 잘 통과하게 됩니다. 따라서 여러분은 `str1` 이 끝났을 때 `str2` 도 끝났는지 확인해볼 필요성이 있습니다.

```
if (*str2 == '\0') return 1;
```

따라서 위 문장을 추가해줌으로써 우리는 `str2` 가 끝이 났는지 확인할 수 있게 됩니다. 만일 `*str2` 가 \0 이 아니라면, 즉 `str2` 가 끝난 것이 아니라면 `str1` 과 `str2` 는 다른 것이 되므로 함수는 0 을 리턴하게 됩니다. 어때요? 간단하죠? 사실 위에서 설명한 4 개의 함수들만 이용하면 문자열을 이용한 왠만한 작업들은 수행이 가능합니다. 이번 강좌는 이쯤게 끝내도록 하겠습니다. 문자열 함수를 이용해서 문자열들을 적절히 가지고 노는 것은 여러분의 몫입니다. 부디 문자열을 가지고 여러가지 재미있는 프로그램을 만들어보았으면 합니다.

생각해보기

문제 1

길이가 최대 100 인 문자열을 하나 입력 받아서 문자열을 역순으로 출력하는 함수를 만들어보세요. (난이도 : 下)예를 들어서 "abcde" 입력 –> "edcba" 출력

문제 2

길이가 최대 100 인 문자열을 입력 받아서 소문자는 대문자로, 대문자는 소문자로 출력하는 함수를 만들어보세요. (난이도 : 下)예를 들어서 "aBcDE" 입력 –> "AbCde" 출력

문제 3

두 개의 문자열을 입력 받아서 같다면 "같다", 다르면 "다르다" 라고 출력하는 함수를 만들어보세요. (난이도 : 下)

문제 4

문자열을 두 개 입력 받아서 먼저 입력받은 문자열에서 나중에 입력받은 문자열의 위치를 검색하는 함수를 만들어보세요. 만일 없다면 -1 을 리턴하고 있다면 그 위치를 리턴합니다. (난이도 : 中)

예를 들어먼저 처음 입력한 것이 `I_am_a_boy` 이고, 나중에 입력한 것이 `am` 이였다면 컴퓨터는 `I_am_a_boy` 에서 `am` 의 위치를 찾는다. 이 경우에는 `am` 의 위치는 2 (처음에서 세번째) 이므로 2 를 리턴한다. 만일 `am` 이라는 문자열이 없다면 -1 을 리턴한다.

문제 5

도서 관리 프로그램을 만들어봅시다. 프로그램에는 다음과 같은 기능들이 구현되어 있어야 합니다. (난이도 : 上)

- 책을 새로 추가하는 기능 (책의 총 개수는 100 권이라 하자. 이 때, 각 책의 정보는 제목, 저자의 이름, 출판사로 한다)
- 책의 제목을 검색하면 그 책의 정보가 나와야 한다.
- 위와 마찬가지로 저자, 출판사 검색 기능이 있어야 한다.
- 책을 빌리는 기능.
- 책을 반납하는 기능

도서 관리 프로젝트

안녕하세요 여러분. 지난번 강좌의 마지막 생각하기 문제를 기억하시나요? 일단 이 강의는 여러분이 그 문제에 대해 충분한 시간 노력해서 생각해 보았다는 것을 가정한 하에 진행하도록 하겠습니다.

지난번에 생각해보기 마지막 문제는 아래와 같았습니다.

도서 관리 프로그램을 만들어봅시다. 프로그램에는 다음과 같은 기능들이 구현되어 있어야 합니다. (난이도 : 上)

- 책을 새로 추가하는 기능 (책의 총 개수는 100 권이라 하자. 이 때, 각 책의 정보는 제목, 저자의 이름, 출판사로 한다)
- 책의 제목을 검색하면 그 책의 정보가 나와야 한다.
- 위와 마찬가지로 저자, 출판사 검색 기능이 있어야 한다.
- 책을 빌리는 기능
- 책을 반납하는 기능

흠. 여러분은 위들 중 얼마나 해결 하셨나요? 저는 개인적으로 여러분이 위들 중에서 적어도 3 개 이상은 했으리라 믿고 싶습니다. 만일 그렇지 않다면 이 강의를 얼른 닫아서 다시 생각해보도록 하세요.

프로그램을 어떻게 만들 것인가?

사실 여러분이 이 문제를 해결했을 때 많은 어려움이 있었을 것이라 생각합니다. 왜냐하면 여러분은 아직까지 조그마한 작업들을 하는 프로그램만을 만들었지 이렇게 거대한(?) 프로그램은 만들어보지 않았기 때문이죠. 이렇게 거대한 프로그램을 만들 때면 이전의 작은 프로그램들을 만들때와는 달리 체계적으로 계획을 세우는 자세가 필요합니다.

이 때, 체계적으로 계획을 세우는 자세란 다음을 모두 생각해보는 것입니다.

1. 이 프로그램은 무슨 작업을 하는가?
2. 과연 이 작업이 꼭 필요한 것인가? (만일 그렇지 않다면 (1) 로 되돌아갑니다)
3. 어떠한 환경에서 프로그램이 작동되는가?
4. 무슨 언어로 개발할 것인가?

정도로 되겠습니다. 일단 1 번의 경우 프로그램을 계획하는 단계에서 가장 중요한 부분 중에 하나입니다. 우리가 만들 프로그램의 경우 "도서 관리 프로그램" 입니다. 이 도서 관리 프로그램에는 정말 도서를 관리하는데 꼭 필요한 기능들 만이 들어가야 되겠지요. 예를 들면 제가 위해서 요구한 것들이지요. 만일 쓸데 없는 작업들을 많이 넣게 되면 프로그램 용량도 커질 뿐더러 개발하는데 드는 시간도 많이 들기 때문에 좋지 않습니다.

2 번의 경우 1 번에서 내가 한 것들을 확인하는 단계입니다. 필요 없는 기능이나 꼭 필요하지 않거나, 아니면 이 프로그램의 목적과 부합하지 않는 작업들의 경우 다시 1 번으로 돌아가 생각해 보아야할 필요성이 있습니다. 우리가 만들어야 할 도서프로그램은 제가 요구한 조건 만을 만들어 주면 충분합니다.

3 번은 우리에게는 큰 문제가 아니지만 실제로 프로그램을 개발하게 되면 상당히 중요한 역할을 차지합니다. 우리가 만드는 프로그램은 Windows 에서 작동될 수 도 있고 Linux 나 MacOS 에서 작동될 수 도 있습니다. 아니면 TV 나 냉장고 아니면 세탁기와 같은 가전 제품에서도 작동될 수 있고 요즘 가장 화제가 되는 iPhone 과 같은 스마트 폰에서 작동될 수 도 있습니다.

우리는 이러한 프로그램의 작동 환경에 맞추어 프로그램을 어떻게 만들어야 될지 고민해야 합니다. 예를 들어 은행의 ATM 에서 작동되는 프로그램은 보안이 최고로 우선이어야 겠죠. 비교적 속도가 느리더라도 말이죠. iPhone 과 같은 스마트 폰에서 작동하는 프로그램은 스마트폰의 사양이 보통 PC 보다 좋지 않으므로 프로그램을 가볍고 빠르게 만들어야 합니다. 우리가 만들 도서 관리 프로그램은 그냥 Windows 에서만 작동되도 충분합니다

4 번은 프로그램을 어떠한 언어 (물론 우리의 경우 무조건 C 이지만...) 로 만들지 결정하는 단계입니다. 세상에서는 수 많은 언어들이 있는데 우리가 지금 배우는 C 말고도 (물론 C 가 가장 기본적이면서도 중요한 언어지요) 각각의 특성을 가지는 언어들이 많습니다. 우리는 이 때 마다 1,2,3 번을 충분히 고려하여 가장 효율적인 언어를 선택해서 프로그램을 만들어야 겠지요. 물론 우리가 만들 도서 프로그램은 C 언어로 만듭니다.

프로그램의 기본 뼈대

자. 그럼 무슨 작업을 하는 프로그램을 만들지 정했으니 이제, 어떠한 방식으로 작동되는지 생각해보도록 합시다. 우리가 만들어야 할 프로그램은 단순히 C 언어 프로그래밍 실력을 키우기 위한 것이기 때문이 굳이 예쁘게 까지 만들 필요는 없을 것 같습니다. 따라서, 이 도서 관리 프로그램은 매우 단순하게, 첫 화면에서 메뉴를 입력 받고 입력 받은 작업을 수행 한 후 다시 메뉴로 돌아오는 것으로 하면 될 것 같습니다.

따라서 프로그램의 기본 뼈대는 아래 처럼 만들 수 있습니다. (물론 여러분이 하신 방법도 좋은 방법일 것입니다. 제 방법은 단순히 참고로만 알아두세요)

```
#include <stdio.h>
int main() {
  int user_choice; /* 유저가 선택한 메뉴*/
```

```
  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");

    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);
    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    }
  }

  return 0;
}
```

성공적으로 컴파일 하면

실행 결과

```
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 1
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
```

```
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 5
```

소스 코드에는 특별히 어려운 부분이 없으나 혹시 다음 문장이 무슨 뜻인지 모를 수 있을 것입니다.

```
while (1)
```

위 말은, 이전에도 이야기 했듯이 컴퓨터는 0 을 거짓, 0 이 아닌 값을 참 으로 판별한다고 말했습니다. 따라서 while 문의 조건이 1 이므로, 다시 말하면 while 문의 조건이 언제나 참이라는 것이지요. 따라서 이 while 문은 무한히 반복되게 됩니다. 우리가 break 를 하지 않을 경우 말이죠.

위 소스 코드에서 중요한 점은 각 작업을 선택할 때 마다 이에 해당하는 '함수' 를 호출한다는 점입니다. 물론, 함수를 반드시 호출할 필요는 없습니다. 그냥 if 문 안에다가 위 작업을 처리하는 코드를 열심히 적어주면 되지요. 하지만 함수를 호출하게 되면 코드를 보기에 상당히 깔끔하며 이해도 잘됩니다.

이 프로그램에 필요한 변수는?

기본적으로 생각해 보아도 책의 제목, 출판사의 이름, 저자의 이름을 저장할 배열이 있어야 합니다. 또한 현재 이 책의 상태 (빌려갔는지, 안 빌려갔는지) 를 표시할 수 있는 배열도 필요합니다. 마지막으로 현재 책의 총 개수가 있어야지만 나중에 책을 새로 추가할 때 배열의 몇 번째 원소에 표시할 지 알 수 있습니다. 따라서 이들을 조합하면 다음과 같이 변수를 선언할 수 있습니다.

```
int user_choice;         /* 유저가 선택한 메뉴 */
int num_total_book = 0; /* 현재 책의 수 */

/* 각각 책, 저자, 출판사를 저장할 배열 생성. 책의 최대 개수는 100 권*/
char book_name[100][30], auth_name[100][30], publ_name[100][30];
/* 빌렸는지 상태를 표시 */
int borrowed[100];
```

이 때 book_name 의 크기가 [100][30] 인 이유는 이전에도 말했듯이 이 도서프로그램에 들어갈 수 있는 책의 최대 개수는 100 권이고, 제목의 크기는 최대 30 자로 제한되기 때문이죠. 나머지 변수들도 마찬가지 입니다. 이 때 borrowed 배열의 경우 원소의 값이 1 이면 빌림, 0 이면 빌리지 않음 이라고 생각하시면 됩니다.

이제 무슨 변수가 필요한지도 알았으니 먼저 1 번 작업, 즉 책을 새로 추가하는 함수를 만들어보도록 합시다. 이름은 add_book 이고 리턴형은 int 로 합시다.

```
/* 책을 추가하는 함수*/
int add_book() {}
```

일단 함수를 만들기 전에 인자로 무엇을 받아야 하는지 생각해봅시다. 책을 추가하려면 책의 이름, 출판사, 저자를 저장할 배열에 대한 포인터를 인자로 받아야 합니다. 그래야지만 이 배열에 새로운 책의 정보를 추가할 수 있지요. 또한 borrowed 배열도 인자로 받아서 기본 설정을 해주어야 합니다. 물론 borrowed 배열의 기본 값은 0, 즉 빌려가지 않음 이겠지요. 마지막으로 num_total_book 도 필요합니다. 왜냐하면 현재 책의 총 수를 알아야 배열의 몇번째 원소에 값을 집어 넣을 지 알게 되기 때문이죠.

이를 종합하여 인자를 만들어보면

```
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {}
```

참고로 팁으로 알려주는 사실은 위와 같이 인자를 쓰는 부분에 엔터를 쳐도 큰 문제는 없습니다. 왜냐하면 C 언어는 위 인자들이 같은 문장에 나열되어 있다고 생각하기 때문이죠. 인자가 길어져서 보기 흉할 때 자주 쓰는 방법 입니다. 자, 그럼 얼른 add_book 함수를 완성시켜봅시다. add_book 함수는 매우 간단합니다.

```
/* 책을 추가하는 함수*/
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {
  printf("추가할 책의 제목 : ");
  scanf("%s", book_name[*num_total_book]);

  printf("추가할 책의 저자 : ");
  scanf("%s", auth_name[*num_total_book]);

  printf("추가할 책의 출판사 : ");
  scanf("%s", publ_name[*num_total_book]);

  borrowed[*num_total_book] = 0; /* 빌려지지 않음*/
  printf("추가 완료! \n");
  (*num_total_book)++;

  return 0;
}
```

저는 위와 같이 하였습니다.

이제 add_book 함수를 이용하기 위해 main 함수의 if (user_choice == 1) 부분에 add_book 함수를 호출하는 코드를 넣어 주시면 됩니다. 아래와 같이 말이지요.

```c
#include <stdio.h>
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
int main() {
  int user_choice;        /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */

  /* 각각 책, 저자, 출판사를 저장할 배열 생성. 책의 최대 개수는 100 권*/
  char book_name[100][30], auth_name[100][30], publ_name[100][30];
  /* 빌렸는지 상태를 표시 */
  int borrowed[100];

  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");

    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);

    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
      add_book(book_name, auth_name, publ_name, borrowed, &num_total_book);
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    }
  }

  return 0;
}
/* 책을 추가하는 함수*/
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {
  printf("추가할 책의 제목 : ");
  scanf("%s", book_name[*num_total_book]);

  printf("추가할 책의 저자 : ");
  scanf("%s", auth_name[*num_total_book]);
```

```
  printf("추가할 책의 출판사 : ");
  scanf("%s", publ_name[*num_total_book]);

  borrowed[*num_total_book] = 0; /* 빌려지지 않음*/
  printf("추가 완료! \n");
  (*num_total_book)++;

  return 0;
}
```

성공적으로 컴파일 했으면

실행 결과

```
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 1
추가할 책의 제목 : The_C_Language
추가할 책의 저자 : Psi
추가할 책의 출판사 : itguru
추가 완료!
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 5
```

어때요. 프로그램이 아주 잘 작동하고 있는 것 같네요. `main` 함수의 `if` 문에서 주의해야 할 점은

```
add_book(book_name, auth_name, publ_name, borrowed, &num_total_book);
```

과 같이 `&` 를 어디에 붙일지 매우 헷갈린다는 것입니다. 기본적으로 배열의 경우, 배열의 이름이 배열의 메모리 상의 시작 주소 이기 때문에 `&` 를 붙일 필요가 없습니다. (정확히 말하면 붙이면

안됩니다). 그러나 num_total_book 과 같은 int 형 변수의 경우 int* 포인터에 주소값을 전달하려면 & 를 이용하여 num_total_book 변수의 주소값을 전달 해주어야 합니다.

또 하나 주의해야 할 부분은 add_book 함수의 원형에서

```
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
```

와 같이 해야 하는데 마지막의 세미 콜론을 빠뜨리게 되면

컴파일 오류

```
error C2085: 'main' : 정식 매개 변수 목록에 없습니다.
error C2143: 구문 오류 : ';'이(가) '{' 앞에 없습니다.
error C2082: 'num_total_book' 정식 매개 변수 재정의
error C2082: 'book_name' 정식 매개 변수 재정의
error C2082: 'auth_name' 정식 매개 변수 재정의
error C2082: 'publ_name' 정식 매개 변수 재정의
error C2082: 'borrowed' 정식 매개 변수 재정의
warning C4047: '함수' : 'int *'의 간접 참조 수준이 'int **'과(와) 다릅니다.
warning C4024: 'add_book' : 형식 및 실제 매개 변수 5의 형식이 서로 다릅니다.
error C2084: 'int add_book(char (*)[30],char (*)[30],char
↪  (*)[30],int *,int *)' 함수에 이미 본문이 있습니다.
```

와 같이 이해하기 힘든 오류의 향연을 맛보게 됩니다.

책 검색하기

자. 이번에는 두 번째 작업, 책을 검색하는 작업을 수행하는 함수를 만들어보기로 합시다. 이 함수의 이름은 serach_book 이라고 합시다. 그렇다면 이 함수는 어떠한 인자를 취해야 될까요? 일단, 단순히 생각해 보아도 book_name, auth_name, publ_name 은 모두 취해야 될 것 같네요. 왜냐하면 우리가 검색을 수행 시, 책 제목에서, 지은이 이름에서, 출판사 이름에서 중 어느 하나를 선택해서 검색할 것이기 때문이죠. 또한 전체 책의 총 개수도 필요합니다. 검색할 때 불필요한 부분은 찾지 않게 하기 말이죠. 결과적으로 함수의 인자는 아래와 같으면 충분하다는 사실을 알 수 있습니다.

```
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book)
```

참고로, 우리가 도서 관리 프로그램에서 만들 "검색" 기능은 우리가 알고 있는 검색 기능과는 살짝 다릅니다. 우리가 흔히 쓰는 검색 기능은 문자열이 비슷하거나 형태를 포함해도 검색 결과에 나타나지만 우리가 만들 검색 기능은 문자열이 완전히 같을 때 나타난다고 합시다. (나중에 업그레이드시 이 부분도 고려해서 하도록 합시다)

그렇다면 머리속에 번뜩이는 생각은 아마,

" 지난번 강좌에서 만들었던 문자열 비교 함수를 이용하면 되겠네!"

그럼, 지난번 강좌에서 문자열 비교 함수 코드를 복사해오겠습니다.

```
int compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
```

이렇게 미리 만들어놓은 소스 코드를 이용하는 것도 매우 중요한 기술중에 하나 입니다. 이미 만든 것을 또 만드느랴 시간을 굳이 낭비할 필요가 없게 되죠.

이번에는 book_search 함수에 어떠한 변수들이 필요할지 생각해봅시다. (굳이 지금 변수들이 뭐가 필요하나 생각을 안해도 됩니다. 저 역시 프로그래밍을 하다가 필요한 변수들이 있으면 그때 그때 추가하는 스타일 입니다. 여기서 집고 넘어가는 것은 기본적으로 무슨 변수들이 필요할 지 생각해 보는 것입니다)

먼저, 사용자의 검색어를 받는 배열이 필요합니다. 따라서 나중에 이 검색어와 책 정보를 비교할 수 있겠지요. 또한 사용자가 어떠한 검색을 할지 (책 제목 검색, 지은이 검색, 출판사 검색) 선택을 할 때도 변수가 필요합니다. 마지막으로 첫번째 책 부터 num_total_book 번째 책 까지 책 정보를 비교하기 현재 몇 번째 책의 정보와 검색어를 비교하고 있는지에 대한 변수가 필요합니다.

그렇다면 아래와 같이 되겠군요.

```
int user_input; /* 사용자의 입력을 받는다. */
int i;
char user_search[30]; /* 사용자가 입력한 검색어 */
```

자 이제, 사용자로 부터 입력을 받아 봅시다.

```
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];

  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);

  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);

  return 0;
}
```

자. 그럼 사용자로 부터 검색어 까지 입력을 받았으니 검색어를 처리하는 일만 남았습니다. 사실 '검색' 이란 말이 거창해 보이지만 우리가 만들 도서 프로그램에서는 상당히 단순합니다. 단순히 compare 함수를 이용해서 책 제목 검색을 했다면, 각 책들의 제목과 user_search 와 비교하면 되는 것이지요. 이 아이디어를 바탕으로 만들면 다음과 같이 됩니다.

```
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];

  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);

  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);

  printf("검색 결과 \n");

  if (user_input == 1) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
```

```
      if (compare(book_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 2) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(auth_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 3) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(publ_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  }

  return 0;
}
```

어때요? 소스가 그다지 어렵지 않죠? 위 소스코드에 대한 해석은 여태까지 강좌를 정말 보았다고 한 사람이라면 이해할 수 있을 것입니다. 그럼, 다시 main 함수에 search_book 함수를 적용시켜 봅시다.

```
#include <stdio.h>
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book);

int compare(char *str1, char *str2);
```

```c
int main() {
  int user_choice;        /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */

  /* 각각 책, 저자, 출판사를 저장할 배열 생성. 책의 최대 개수는 100 권*/
  char book_name[100][30], auth_name[100][30], publ_name[100][30];
  /* 빌렸는지 상태를 표시 */
  int borrowed[100];

  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");

    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);

    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
      add_book(book_name, auth_name, publ_name, borrowed, &num_total_book);
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
      search_book(book_name, auth_name, publ_name, num_total_book);
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    }
  }

  return 0;
}
/* 책을 추가하는 함수*/
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {
  printf("추가할 책의 제목 : ");
  scanf("%s", book_name[*num_total_book]);

  printf("추가할 책의 저자 : ");
  scanf("%s", auth_name[*num_total_book]);

  printf("추가할 책의 출판사 : ");
  scanf("%s", publ_name[*num_total_book]);
```

```
  borrowed[*num_total_book] = 0; /* 빌려지지 않음*/
  printf("추가 완료! \n");
  (*num_total_book)++;

  return 0;
}
/* 책을 검색하는 함수 */
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];

  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);

  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);

  printf("검색 결과 \n");

  if (user_input == 1) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(book_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 2) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(auth_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
```

```
    }

  } else if (user_input == 3) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(publ_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  }

  return 0;
}
int compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 2
```

```
어느 것으로 검색 할 것인가요?
1. 책 제목 검색
2. 지은이 검색
3. 출판사 검색
2
검색할 단어를 입력해주세요 : Psi
검색 결과
번호 : 0 // 책 이름 : The_C_Language // 지은이 : Psi // 출판사 : itguru
번호 : 1 // 책 이름 : AdvancedCProgramming // 지은이 : Psi // 출판사 :
↪  modoocode
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 5
```

위와 같이 아주 잘 작동함을 알 수 있습니다.

아마도 위 도서 프로그램에서는 검색하기 기능이 가장 어려운 것 같네요. 나머지 기능들은 정말로 단순합니다. 3 번째 기능인 "책을 빌리기" 는 단순히 사용자가 빌리려는 책의 번호를 입력하면 borrowed 배열의 책 번호에 위치한 원소의 값을 0 에서 1 로 바꾸어주면 됩니다. 왜냐하면 borrowed 가 1 이면 빌려진 것, 0 이 면 안 빌려진 것 이기 때문이죠. 마찬가지로 책을 반납하는 기능도 만들 수 있습니다.

3, 4 번 기능

먼저, 3 번 기능 부터 만들어봅시다. 함수 이름은 borrow_book 으로 합시다.

```c
int borrow_book(int *borrowed) {
  /* 사용자로 부터 책번호를 받을 변수*/
  int book_num;

  printf("빌릴 책의 번호를 말해주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &book_num);

  if (borrowed[book_num] == 1) {
    printf("이미 대출된 책입니다! \n");
```

```
  } else {
    printf("책이 성공적으로 대출되었습니다. \n");
    borrowed[book_num] = 1;
  }

  return 0;
}
```

사실 위 함수는 매우 매우 간단하므로 특별히 설명할 것은 없습니다. 다만 주의할 점은 책이 이미 대출되어 있는 경우에도 처리를 잘 해주어야 한다는 점입니다. 책이 대출되어 있을 경우 "이미 대출된 책입니다" 라는 메세지를 표시하고 대출을 시키면 안됩니다. 자, 그럼 이제 위 함수를 `main` 함수에 넣어 작동시켜 봅시다.

```
#include <stdio.h>
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book);

int compare(char *str1, char *str2);
int borrow_book(int *borrowed);

int main() {
  int user_choice;        /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */

  /* 각각 책, 저자, 출판사를 저장할 배열 생성. 책의 최대 개수는 100 권*/
  char book_name[100][30], auth_name[100][30], publ_name[100][30];
  /* 빌렸는지 상태를 표시 */
  int borrowed[100];

  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");

    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);

    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
      add_book(book_name, auth_name, publ_name, borrowed, &num_total_book);
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
      search_book(book_name, auth_name, publ_name, num_total_book);
```

```
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
      borrow_book(borrowed);
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    }
  }

  return 0;
}
/* 책을 추가하는 함수*/
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {
  printf("추가할 책의 제목 : ");
  scanf("%s", book_name[*num_total_book]);

  printf("추가할 책의 저자 : ");
  scanf("%s", auth_name[*num_total_book]);

  printf("추가할 책의 출판사 : ");
  scanf("%s", publ_name[*num_total_book]);

  borrowed[*num_total_book] = 0; /* 빌려지지 않음*/
  printf("추가 완료! \n");
  (*num_total_book)++;

  return 0;
}
/* 책을 검색하는 함수 */
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];

  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);

  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);

  printf("검색 결과 \n");

  if (user_input == 1) {
    /*
```

```
    i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(book_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 2) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(auth_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 3) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(publ_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  }

  return 0;
}
int compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
```

```
  }

  if (*str2 == '\0') return 1;

  return 0;
}
int borrow_book(int *borrowed) {
  /* 사용자로 부터 책번호를 받을 변수*/
  int book_num;

  printf("빌릴 책의 번호를 말해주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &book_num);

  if (borrowed[book_num] == 1) {
    printf("이미 대출된 책입니다! \n");
  } else {
    printf("책이 성공적으로 대출되었습니다. \n");
    borrowed[book_num] = 1;
  }

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 3
빌릴 책의 번호를 말해주세요
책 번호 : 0
책이 성공적으로 대출되었습니다.
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
```

```
5. 프로그램 종료
당신의 선택은 : 5
```

위와 같이 책이 잘 대출됨을 알 수 있습니다.

마찬가지 아이디어를 이용해서 책을 반납하는 함수를 만들어봅시다. 함수의 이름은 return_book 으로 합시다. 이 역시 borrow_book 과 하는 일이 거의 똑같으므로 설명을 생략하도록 하겠습니다.

```c
int return_book(int *borrowed) {
  /* 반납할 책의 번호 */
  int num_book;

  printf("반납할 책의 번호를 써주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &num_book);

  if (borrowed[num_book] == 0) {
    printf("이미 반납되어 있는 상태입니다\n");
  } else {
    borrowed[num_book] = 0;
    printf("성공적으로 반납되었습니다\n");
  }

  return 0;
}
```

역시 간단하군요. borrow_book 함수의 거의 똑같습니다. 이제, 이 함수를 main 함수에 넣어봅시다.

```c
#include <stdio.h>
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book);

int compare(char *str1, char *str2);
int borrow_book(int *borrowed);
int return_book(int *borrowed);

int main() {
  int user_choice;        /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */

  /* 각각 책, 저자, 출판사를 저장할 배열 생성. 책의 최대 개수는 100 권*/
  char book_name[100][30], auth_name[100][30], publ_name[100][30];
  /* 빌렸는지 상태를 표시 */
  int borrowed[100];
```

```
  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");

    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);

    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
      add_book(book_name, auth_name, publ_name, borrowed, &num_total_book);
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
      search_book(book_name, auth_name, publ_name, num_total_book);
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
      borrow_book(borrowed);
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
      return_book(borrowed);
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    }
  }

  return 0;
}
/* 책을 추가하는 함수*/
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {
  printf("추가할 책의 제목 : ");
  scanf("%s", book_name[*num_total_book]);

  printf("추가할 책의 저자 : ");
  scanf("%s", auth_name[*num_total_book]);

  printf("추가할 책의 출판사 : ");
  scanf("%s", publ_name[*num_total_book]);

  borrowed[*num_total_book] = 0; /* 빌려지지 않음*/
  printf("추가 완료! \n");
  (*num_total_book)++;

  return 0;
}
/* 책을 검색하는 함수 */
```

```
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];

  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);

  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);

  printf("검색 결과 \n");

  if (user_input == 1) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(book_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 2) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(auth_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 3) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를
    사용자가 입력한 검색어와 비교하고 있다.
```

```
    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(publ_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  }

  return 0;
}
int compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
int borrow_book(int *borrowed) {
  /* 사용자로 부터 책번호를 받을 변수*/
  int book_num;

  printf("빌릴 책의 번호를 말해주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &book_num);

  if (borrowed[book_num] == 1) {
    printf("이미 대출된 책입니다! \n");
  } else {
    printf("책이 성공적으로 대출되었습니다. \n");
    borrowed[book_num] = 1;
  }

  return 0;
}
int return_book(int *borrowed) {
  /* 반납할 책의 번호 */
  int num_book;

  printf("반납할 책의 번호를 써주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &num_book);

  if (borrowed[num_book] == 0) {
```

```
    printf("이미 반납되어 있는 상태입니다\n");
  } else {
    borrowed[num_book] = 0;
    printf("성공적으로 반납되었습니다\n");
  }

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 3
빌릴 책의 번호를 말해주세요
책 번호 : 0
책이 성공적으로 대출되었습니다.
도서 관리 프로그램
메뉴를 선택하세요
1. 책을 새로 추가하기
2. 책을 검색하기
3. 책을 빌리기
4. 책을 반납하기
5. 프로그램 종료
당신의 선택은 : 4
반납할 책의 번호를 써주세요
책 번호 : 0
성공적으로 반납되었습니다
```

위와 같이 아주 잘 작동됨을 알 수 있습니다.

아. 그럼 마침내 우리는 도서 관리 프로그램을 완성하였습니다!! 처음에 막연하게 도서 관리 프로그램을 만들라고 하니까 상당히 막연해 보였는데 하나 하나 조금씩 해보니 금새 만들게 되었습니다. 뿐만 아니라 함수를 이용해서 소스 코드의 가독성도 높여주었습니다.

뿐만 아니라 여러분이 도서 관리 프로그램을 만들면서 느끼게 될 점은 **주석의 유용함** 일 것입니다. 만일 우리가 주석 하나 없이 단순하게 프로그램을 만들었다면 나중에.. 한 1 달 뒤에 이 소스 코드를 다시 보게 된다면 "이 변수가 뭐지? 이 부분은 뭐하는 것이지?" 라는 생각이 들 것입니다. 저는 심지어 주석이 없이 프로그래밍 한 경우 어제 한 코드도 이해 못하는 경우가 있었습니다. 하지만 주석이 있다면 정말 아무리 오래 전에 만든 프로그램이여도 소스를 보면서 손쉽게 이해해 나갈 수 있습니다.

그럼 여기서 이번 강좌를 마치도록 하겠습니다.

생각해보기

문제 1

위 프로그램을 다 지우고 다시 만들어보자. 물론 소스가 정확히 일치하지 않아도 된다. 기능만 동일하면 된다. (난이도 : 下)

문제 2

`search_book` 함수는 살짝 지저분한 편이다. 다른 함수를 제작하여 조금 간추릴 수 있겠는가? (난이도 : 中下)

문제 3

`search_book` 함수를 조금 개량하여 빌려진 책은 검색결과 출력되지 않게 하거나, "대출됨" 이라는 문구가 출력되게 해보자. (난이도 : 下)

문제 4

`search_book` 함수를 개량하여 특정한 검색어를 입력했을 때 그 검색어를 포함하는 문자열도 검색되게 해보자. (난이도 : 中)

예를 들어 책 제목이 "learnCfast", "learningC", "whatisC?" 일 때, learn 를 검색하면 "learnCfast" 와 "learningC" 가 나온다. 왜냐하면 이들은 모두 "learn 라는 문자열을 포함하고 있기 때문이다.

14

구조체 (struct)

안녕하세요 여러분. 잘 지내셨는지요? 제가 요즘에 강좌를 올리는 틈틈히 C 레퍼런스를 정리하고 있습니다. 레퍼런스라 하면, 일종의 백과사전 같은 것으로 여러분들이 궁금한 함수들이 있다면 찾아볼 수 있게 해놓았습니다. 아직 일부 함수들 밖에 올리지는 못했지만 그래도 그 양이 꽤 되니 읽어 보시는 것이 좋을 듯 합니다.

특히, 화면에서 입력을 받는 함수는 아직 `scanf` 와 `getchar` 밖에 보지 못했지만 `fgets, gets` 등이 있고, 화면에 출력하는 함수는 `fputs`, `puts`, `putchar` 등등 매우 많습니다. 뿐만 아니라 우리가 `scanf` 함수나 `printf` 함수를 여태까지 써오면서 사용하지 못했던 기능이 수없이 많은데 이들 모두 레퍼런스에 잘 정리되어 있으니 참조하시기 바랍니다.

여러분은 지금 Sims 라는 게임을 만들고 있습니다. Sims 라는 게임은 워낙 유명하지만, 그래도 뭔지 설명 해보자면 사람을 육성(?) 하는 게임 입니다. 만일 게임하는 유저가 사람을 한 명 추가했다고 합시다. 그렇다면 이를 프로그램 상에서 어떻게 저장할까요?

여러분의 머리속에는 "음, 그럼 이전에 만들어놓은 문자열 배열의 `i` 번째 원소에 이름을 등록하고, `int` 형 배열의 `i` 번째 원소에 나이를 등록하고, 성격은..." 과 같이 생각할 것입니다. 맞아요. 이렇게 한 방법은 우리가 지난 강좌에서 도서 관리 프로그램을 만들 때 사용했던 방법이지요. `i` 번째 책에 대한 정보는 `book_name[i - 1], auth_name[i - 1], publ_name[i - 1] , borrow[i - 1]` (`i` 번째 이므로, 원소는 `[i - 1]` 이겠지요) 배열에 넣어서 보관하였습니다.

그런데 말이죠. 위 방법에는 살짝 문제점이 있었습니다. 책의 정보를 수정하기 위해서 함수에 인자로 전달할 때 상당히 불편하다는 사실입니다. 한 번 도서 관리 프로그램의 새로운 책을 추가하는 함수였던 add_book 함수의 원형을 가져와보았습니다.

```
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
```

헐. num_total_book 을 빼더라도 인자가 너무나 깁니다. 그래도 도서 관리 프로그램은 봐줄만 했죠. 하지만 우리가 만들게 될 Sims 는 다릅니다. 일단 사람 한 명에는 수없이 많은 정보가 있습니다. 예를 들어, 이름, 나이, 직업, 성격 (외향적, 내향적, 사교적, ... 들을 모두 수치화 시켜서 보관한다), 직업, 재산, 가족 관계 등등 수없이 많은 정보가 있습니다. 책과는 완전히 다르지요. 우리가 사람의 정보를 수정하기 위해 함수를 호출할 때 마다 이렇게 무지막지 하게 많은 정보들을 인자로 전달하려면 손가락이 빠질 뿐더러 눈도 매우 아프게 됩니다.

우리가 배열을 배웠기 전의 모습을 생각해봅시다. 배열을 배우기 전에, 예를 들어 10 명의 학생의 점수를 보관하기 위해 10 개의 변수를 선언해서 각각에 보관했어야 했습니다. 하지만, 배열을 배운 이후는 어떨까요? 10 개의 변수를 한꺼번에 배열로 처리하여 배열의 각각의 원소를 손쉽게 다루기 위해서였습니다. 즉 `int arr[10];` 이라 한다면 `int` 형 변수 10 개를 쉽게 다룰 수 있는 것입니다. 뿐만 아니라 함수에 `int` 형 변수를 전달할 때, `int` 형 변수 10 개를 일일히 전달하려면

```
int func(int a, int b, int c, .....(생략)...)
```

와 같이 해야 합니다. 그런데 `int` 형 변수 10 개를 `arr` 을 이용하여 쉽게 전달할 수 있습니다. 아래와 같이요.

```
int func(int *arr);
```

그렇다면 위 배열과 같은 논리가 여기에도 적용될 수 있지 않을까요? 원소의 크기가 제각각인 배열을 만드는 것입니다.한 사람에 대한 정보를 한 개의 배열에 저장하는 것입니다! 첫번째 원소는 `int` 로 나이를 보관하고, 두 번째 원소는 `char [30]` 으로 이름을 보관하는 것입니다.

정말 괜찮은 아이디어입니다. 하지만 C 언어에서는 배열의 원소의 타입은 모두 동일해야 합니다. 다시말해 동일한 배열에서 어떤 원소는 `char` 이고 어떤 원소는 `int` 일 수 없다는 것이죠. 다행스럽게도 C 언어에서는 배열로 해결하지 못하는 문제를 **구조체** 를 이용하여 해결할 수 있었습니다.

```
/* 구조체의 도입*/
#include <stdio.h>
struct Human {
  int age;    /* 나이 */
  int height; /* 키 */
  int weight; /* 몸무게 */
```

```
};            /* ; 붙이는 것 주의하세요 */
int main() {
  struct Human Psi;

  Psi.age = 99;
  Psi.height = 185;
  Psi.weight = 80;

  printf("Psi 에 대한 정보 \n");
  printf("나이   : %d \n", Psi.age);
  printf("키     : %d \n", Psi.height);
  printf("몸무게 : %d \n", Psi.weight);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
Psi 에 대한 정보
나이   : 99
키     : 185
몸무게 : 80
```

아마 여러분은 위 소스 코드에서 여러가지 새로운 것들을 보실 수 있으셨을 겁니다. 아마 이 강좌가 끝날 즈음에는 위 사실들을 자유롭게 다룰 수 있게 되니 크게 걱정 안하셔도 됩니다. 일단 구조체를 정의한 부분 부터 살펴 봅시다. 직관적으로 아래의 부분과 같다는 사실을 알 수 있습니다.

```
struct Human {
  int age;    /* 나이 */
  int height; /* 키 */
  int weight; /* 몸무게 */
};            /* ; 붙이는 것 주의하세요 */
```

앞서 말했듯이 구조체는 "각 원소의 타입이 제각각인 배열" 이라고 말했습니다. 이 때문에 배열에서는 배열의 타입만으로 모든 원소의 타입을 알 수 있었지만 (예를 들어 `int array[100]` 이면 `array` 의 모든 원소의 타입은 `int` 형 이다)

구조체는 그렇지 않습니다. 따라서 구조체는 정의할 때 모든 원소의 타입을 명시해 주어야 합니다. 위와 같이 말이죠. 이 `Human` 이라는 이름의 구조체는 3 개의 멤버를 가지고 있는데 (보통 구조체에서는 원소 보다는 **멤버(member)** 라고 부릅니다) 각각의 멤버는 `int age`, `int height`, `int weight` 로 3 개가 있습니다.

구조체의 일반적인 정의는 아래와 같습니다.

```
struct 구조체이름 {
  멤버들..예를 들면 char str[10];
  int i;
}; /* 마지막에 꼭 ; 를 붙인다. */
```

다음은 main 함수내부를 살펴 볼 것입니다.

```
struct Human Psi;
```

위와 같이 Human 이라는 구조체의 구조체 변수 Psi 를 정의하였습니다. 여기서 놀라운 점은 struct Human 이라는 것이 우리가 마치 int 형 변수를 정의할 때 int 를 쓰는 것과 같이 사용되었다는 것입니다. 아무튼 이처럼 Psi 를 정의하고 나면,Psi 의 타입은 struct Human, 즉 Human 구조체가 됩니다. int a 했을 때 a 의 타입이 int 인 것처럼 말이지요.

그렇다면 배열에서 [] 를 이용해서 원소에 접근하듯이, 구조체에서도 멤버에 접근할 방법이 있어야 겠죠? C 언어에서는 . 을 이용하여 원소에 접근할 수 있습니다. 예를 들어서, Psi 의 height 멤버에 접근하려면 Psi.height 라고 하면 됩니다. 이는 마치 배열에서 arr[3] 과 같이 원소에 접근하는 것과 동일한 것입니다. 다만 구조체는 . 을 이용하고, 멤버가 무엇인지 특별히 명시해주어야 하는 것만 빼고요.

```
Psi.age = 99;
Psi.height = 185;
Psi.weight = 80;
```

따라서, 위 작업은 Psi 라는 구조체의 각 멤버에 값을 대입하는 것입니다. 이는 마치 배열에서 arr[1] = 99, arr[2] = 185 와 같이 하는 것과 동일합니다.

지금 구조체를 처음 배워서 살짝 이해가 안되는 것도 있고 무언가 혼동되는 것이 있을 것입니다. 그래도 구조체가 무엇인지는 감이 대충 오지 않나요? 이제, 다음 예제를 살펴 보아서 구조체가 뭔지 감을 확 잡아보도록 합시다.

```
/* 구조체 예제 2 */
#include <stdio.h>
char copy_str(char *dest, const char *src);
struct Books {
  /* 책 이름 */
  char name[30];
  /* 저자 이름 */
  char auth[30];
  /* 출판사 이름 */
  char publ[30];
  /* 빌려 졌나요? */
```

```
  int borrowed;
};
int main() {
  struct Books Harry_Potter;

  copy_str(Harry_Potter.name, "Harry Potter");
  copy_str(Harry_Potter.auth, "J.K. Rolling");
  copy_str(Harry_Potter.publ, "Scholastic");
  Harry_Potter.borrowed = 0;

  printf("책 이름 : %s \n", Harry_Potter.name);
  printf("저자 이름 : %s \n", Harry_Potter.auth);
  printf("출판사 이름 : %s \n", Harry_Potter.publ);

  return 0;
}
char copy_str(char *dest, const char *src) {
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }

  *dest = '\0';

  return 1;
}
```

성공적으로 컴파일 했다면

실행 결과

```
책 이름 : Harry Potter
저자 이름 : J.K. Rolling
출판사 이름 : Scholastic
```

일단, 저는 지지난 강좌에서 만들었던 `copy_str` 함수를 가져왔습니다. 이 함수는 문자열을 `src` 에서 `dest` 로 복사하는 함수이지요. 이렇게 이미 썼던 것을 활용하는 것은 상당히 시간도 절약되고 편리한 방법 중에 하나 입니다.

먼저, 구조체를 정의한 부분 부터 살펴 보도록 합시다.

```
struct Books {
  /* 책 이름 */
  char name[30];
  /* 저자 이름 */
  char auth[30];
```

```
  /* 출판사 이름 */
  char publ[30];
  /* 빌려 졌나요? */
  int borrowed;
};
```

흥미로운 점은 이 Book 구조체가 우리가 이전에 만들었던 도서 관리 프로그램을 쏙 빼닮았다는 것이죠. 그 때 에는 각 책을 배열의 한 개의 원소로 표현했는데, 책 이름의 경우 name[100][30] 의 한 문자열 name[i] 으로, (여기서 i 는 임의의 수), 빌려 졌는지에 대한 유무의 경우, borrowed [100] 의 한 원소 borrowed[i] 로 표현했었죠. 하지만 구조체를 이용하면 책의 각각의 정보를 따로 따로 배열에 정의할 필요가 없게 됩니다. main 함수를 살펴보면

```
struct Books Harry_Potter;

copy_str(Harry_Potter.name, "Harry Potter");
copy_str(Harry_Potter.auth, "J.K. Rolling");
copy_str(Harry_Potter.publ, "Scholastic");
Harry_Potter.borrowed = 0;
```

먼저 우리는 Harry_Potter 라는 struct Books 의 구조체 변수를 만들었습니다. 자, 그럼 Harry_Potter 의 각 멤버에 값을 대입해야 겠죠? 먼저 책의 이름, 즉 Harry_Potter.name 에 "Harry Potter" 를, 마찬가지로 저자 이름과 출판사에도 모두 대입합니다. 마지막으로 빌렸는지 안빌렸는지에 대한 유무 확인을 위한 Harry_Potter.borrowed 에도 0 을 넣어주어야 합니다.

그런데 말이죠. borrowed 멤버의 값은 처음에 언제나 0 으로 설정되어있습니다. 그렇다면 굳이 매번 책을 새로 등록할 때 마다 borrowed = 0 을 해줄 필요 없이 구조체 자체 에서 바꿔버리면 안될까요?

한 번 구조체 정의 부분을

```
struct Books {
  /* 책 이름 */
  char name[30];
  /* 저자 이름 */
  char auth[30];
  /* 출판사 이름 */
  char publ[30];
  /* 빌려 졌나요? */
  int borrowed = 0;
};
```

로 바꿔서 컴파일 해보세요. 과연..?잘 되리라 기대했지만 아래와 같은 오류의 향연을 보실 수 있습니다.

컴파일 오류

```
error C2143: 구문 오류 : ';'이(가) '=' 앞에 없습니다.
error C2059: 구문 오류 : '='
error C2059: 구문 오류 : '}'
error C2079: 'Harry_Potter'은(는) 정의되지 않은 struct 'Books'을(를)
 ↪  사용합니다.
error C2224: '.name' 왼쪽에는 구조체/공용 구조체 형식이 있어야 합니다.
error C2198: 'copy_str' : 호출에 매개 변수가 너무 적습니다.
error C2224: '.auth' 왼쪽에는 구조체/공용 구조체 형식이 있어야 합니다.
error C2198: 'copy_str' : 호출에 매개 변수가 너무 적습니다.
error C2224: '.publ' 왼쪽에는 구조체/공용 구조체 형식이 있어야 합니다.
error C2198: 'copy_str' : 호출에 매개 변수가 너무 적습니다.
error C2224: '.borrowed' 왼쪽에는 구조체/공용 구조체 형식이 있어야 합니다.
error C2224: '.name' 왼쪽에는 구조체/공용 구조체 형식이 있어야 합니다.
error C2224: '.auth' 왼쪽에는 구조체/공용 구조체 형식이 있어야 합니다.
error C2224: '.publ' 왼쪽에는 구조체/공용 구조체 형식이 있어야 합니다.
```

도대체 이게 뭔일인가요?

사실 위 처럼 나온 이유는 간단합니다. 구조체의 정의에서는 변수를 초기화 할 수 없기 때문입니다. 그냥, 받아 들여주세요. 구조체 정의 내부에서는 변수를 초기화 할 수 없다고 말이죠. 특히, 위와 같이 실수를 할 경우 찾기도 잘 어렵고 오류들도 엉뚱한 것들만 나오기 때문에 위와 같은 실수를 조심하는 것이 아주 중요합니다.

```c
/* 구조체 예제*/
#include <stdio.h>
struct Books {
  /* 책 이름 */
  char name[30];
  /* 저자 이름 */
  char auth[30];
  /* 출판사 이름 */
  char publ[30];
  /* 빌려 졌나요? */
  int borrowed;
};
int main() {
  struct Books book_list[3];
  int i;

  for (i = 0; i < 3; i++) {
    printf("책 %d 정보 입력 : ", i);
    scanf("%s%s%s", book_list[i].name, book_list[i].auth, book_list[i].publ);
```

```
    book_list[i].borrowed = 0;
  }

  for (i = 0; i < 3; i++) {
    printf("------------------------------ \n");
    printf("책 %s 의 정보\n", book_list[i].name);
    printf("저자 : %s \n", book_list[i].auth);
    printf("출판사 : %s \n", book_list[i].publ);

    if (book_list[i].borrowed == 0) {
      printf("안 빌려짐\n");
    } else {
      printf("빌려짐 \n");
    }
  }
  return 0;
}
```

성공적으로 컴파일 되었다면

실행 결과

```
책 0 정보 입력 : ChewingC Psi itguru
책 1 정보 입력 : ChewingCPP Psi ModooCode
책 2 정보 입력 : asdf asdf as
------------------------------
책 ChewingC 의 정보
저자 : Psi
출판사 : itguru
안 빌려짐
------------------------------
책 ChewingCPP 의 정보
저자 : Psi
출판사 : ModooCode
안 빌려짐
------------------------------
책 asdf 의 정보
저자 : asdf
출판사 : as
안 빌려짐
```

먼저 구조체의 정의에 대한 부분은 생략하고 바로 `main` 함수 부터 이야기 하겠습니다.

```
struct Books book_list[3];
```

일단 위 문장을 보았을 때 어떤 분들은 이해가 잘 되지만 어떤 분들은 이해가 안될 수 도 있을 것이니 다시 한 번 설명하겠습니다. 이전에도 말했듯이 `int arr[3]` 에서 `int` 가 하나의 타입이듯이, `struct Books` 가 하나의 타입으로 생각하면 됩니다. 그런데, `int arr[3]` 을 하면 `arr` 에 `int` 형 원소가 3 개 만들어지듯이, `book_list` 배열에는 `struct Books` 형의 변수가 3 개 만들어지는 것이지요.

```
for (i = 0; i < 3; i++) {
  printf("책 %d 정보 입력 : ", i);
  scanf("%s%s%s", book_list[i].name, book_list[i].auth, book_list[i].publ);
  book_list[i].borrowed = 0;
}
```

이제, `for` 문을 살펴봅시다. `scanf` 함수로 `book_list` 의 `i` 원소의 `name, auth, publ` 멤버에 문자열을 입력받고 있는 모습을 볼 수 있습니다 또한 `borrowed` 의 값도 0 으로 초기화 해주고 있습니다.

```
for (i = 0; i < 3; i++) {
  printf("------------------------------ \n");
  printf("책 %s 의 정보\n", book_list[i].name);
  printf("저자 : %s \n", book_list[i].auth);
  printf("출판사 : %s \n", book_list[i].publ);

  if (book_list[i].borrowed == 0) {
    printf("안 빌려짐\n");
  } else {
    printf("빌려짐 \n");
  }
}
```

입력을 다 받고 나면 `for` 문에서 `book_list` 의 각 원소의 멤버들을 출력해줍니다. 특히 `borrowed` 값이 0 이면 "안빌려짐", 0 이 아니면 "빌려짐" 이 출력되는데 위의 경우 0 으로 값을 설정해 놓고 값을 바꾸는 부분이 없으므로 언제나 안 빌려짐이 출력됩니다. 어때요? 간단하죠?

구조체 포인터

으음... 위 파란색 제목만 보고도 눈살을 찌부리는 분들이 있을 지도 모릅니다. 한동안 포인터에게서 벗어난 줄 알았는데 또 등장하는거냐! 하지만 구조체 포인터, 말그대로 구조체를 가리키는 포인터 역시 잘만 이해하면 정말로 아무 것도 아닌 것이 됩니다. 오히려 나중엔 "내가 왜 여기서 겁먹었지?" 라는 생각이 들 정도로요.

```c
/* 구조체 포인터 */
#include <stdio.h>
struct test {
  int a, b;
};
int main() {
  struct test st;
  struct test *ptr;

  ptr = &st;

  (*ptr).a = 1;
  (*ptr).b = 2;

  printf("st 의 a 멤버 : %d \n", st.a);
  printf("st 의 b 멤버 : %d \n", st.b);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
st 의 a 멤버 : 1
st 의 b 멤버 : 2
```

먼저 구조체 포인터에 대해 이야기 하기 전에 확실히 짚고 넘어가야 할 것이 있습니다. 여태까지 누누히 이야기 하였지만 `struct test` 역시 하나의 형 (타입) 이라는 것입니다. 위의 예제들의 `struct Human` 이나 `struct Book` 역시 하나의 타입이였지요.

즉, 구조체는 한 개의 타입을 창조하는 것과 마찬가지라는 것입니다. 마치 `int` 나 `char` 처럼 말이지요. 그런데 이러한 타입들을 가리킬 때 우리가 포인터를 어떻게 사용했나요? 바로, `int *` 나 `char *` 로 사용했습니다. 구조체도 마찬가지 입니다.

```c
struct test st;
struct test *ptr;
```

위의 두 번째 문장과 같이 `struct test *ptr`, 즉 "struct `test` 형을 가리키는 포인터 ptr" 을 정의한 것이지요. 여기서 주의해야 할 점은 `ptr` 은 절대로 **구조체가 아니라는 것**입니다. `ptr` 역시 다른 모든 포인터 처럼 4 바이트의 공간을 차지하는 것입니다. (물론 컴퓨터 마다 다를 수 있지만

아마 여러분이 사용하는 컴퓨터는 십중 팔구일 것입니다)[1)]

```
ptr = &st;
```

그리고 위와 같이 ptr 에 str 의 주소값을 집어 넣습니다. 그런데 눈치가 조금 빠르신 분들은 다음과 같이 질문할 수 있습니다.

"아까 구조체는 단순히 원소의 크기가 제각각인 배열이라면서요? 그러면 구조체도 배열처럼 변수의 이름이 그 주소값이 되어야 하는 것 아닌가요? 다시 말해 우리가 int arr[100]; 을 정의했다면 이를 가리키는 포인터를 정의할 때 int *ptr = arr 이라고 하지 int *ptr = &arr 이라 하지 않잖아요?"

상당히 좋은 질문 입니다. 하지만 조금 아래에 보면 구조체 변수의 이름은 역할이 살짝 다르다는 것을 알게 됩니다. 그냥 보통 변수 처럼, (그래서 구조체 변수라 부르지, 구조체 '배열' 이라고 부르지 않잖아요) & 를 붙여 구조체가 정의된 메모리의 주소값을 얻어온다고 생각해주세요.

이제 ptr 은 구조체 st 를 가리키는 포인터가 됩니다.

```
(*ptr).a = 1;
(*ptr).b = 2;
```

그럼 ptr 이 가리키는 구조체의 멤버의 값을 변경하는 부분을 살펴 봅시다. 일단 여러분은 (*ptr) 이라는 부분이 st 라는 것과 동일하다는 사실을 알 수 있습니다. 왜냐하면 ptr 이 st 를 가리키고 있기 때문이죠. 따라서 (*ptr).a = 1 은 st.a = 1 과 완전히 100% 동일한 문장임을 알 수 있습니다. 그 아랫 줄도 마찬가지이죠. (*ptr).b = 2 도 st.b = 2 와 정확히 일치하는 문장입니다. 따라서 아래 printf 문에서 st.a 의 값은 1, st.b 의 값은 2 가 출력된 것입니다.

그런데 말이죠. 굳이 괄호를 쳐 주어야 하나요? 그냥 *ptr.a = 1 이라 하면 무엇이 문제이길래 그런 것인가요?

(*ptr).a = 1; 을 *ptr.a = 1; 로 바꿔서 컴파일 해봅시다.

그렇다면 아래의 오류들을 만나실 수 있을 것입니다.

컴파일 오류

```
error C2231: '.a' : 왼쪽 피연산자가 'struct'을(를) 가리킵니다. '->'를
↪  사용하십시오.
error C2100: 간접 참조가 잘못되었습니다.
```

1) 이 강좌는 2010년에 제작되었습니다. 2018년 지금 현재 64비트 시스템을 사용하시는 분들의 경우 포인터는 8 바이트를 차지합니다.

도대체 왜 발생한 것일까요?이에 대해 답하기 전에 연산자 우선 순위 표를 먼저 살펴봅시다.

1	( ) [ ] -> .	왼쪽 우선
2	! ~ ++ -- + -(부호) *(포인터) & sizeof 캐스트	오른쪽 우선
3	*(곱셈) / %	왼쪽 우선
4	+ -(덧셈, 뺄셈)	왼쪽 우선
5	<< >>	왼쪽 우선
6	< <= > >=	왼쪽 우선
7	== !=	왼쪽 우선
8	&	왼쪽 우선
9	^	왼쪽 우선
10	|	왼쪽 우선
11	&&	왼쪽 우선
12	||	왼쪽 우선
13	? :	오른쪽 우선
14	= 복합대입	오른쪽 우선
15	,	왼쪽 우선

가장 맨 위를 보면 `.` 이라고 되있는 것을 볼 수 있습니다. 찾았나요? 여기서 `.` 은 구조체의 멤버를 지칭할 때 사용하는 `.` 을 의미하는 것입니다. `(*ptr).a` 에서 사용된 `.` 을 말하지요. 그 바로 아래 행을 보면 `*(포인터)` 라고 써있는 것이 있습니다. `(*ptr).a` 에서의 `*` 를 말하는 것이지요. 여기서 주목해야 할 점은 `.` 이 `*` 보다 우선순위가 높다는 것입니다.

따라서, `*ptr.a` 를 사용하게 되면 `ptr.a` 를 먼저 실행한 후, 그 값에 `*` 를 한 것에 2 가 들어가게 됩니다. 즉 `*ptr.a` 는 `*(ptr.a)` 와 동일한 문장인 것이지요. 그런데 위에서도 말했지만 `ptr` 은 단순히 포인터에 불과합니다. `ptr` 은 절대로 구조체가 아니라는 것이지요. 그런데 구조체가 아닌 것의 있지도 않는 `a` 라는 멤버에 접근하라니 컴파일 시에 오류가 발생하는 것입니다.

결과적으로 구조체 포인터를 사용해서 멤버에 접근하려면 `(*ptr).a` 와 같이 언제나 괄호로 감싸주어야 됩니다. 상당히 귀찮은 일이 아닐 수 없습니다. 하지만 똑똑한 C 프로그래머들은 이 문제를 해결하기 위해 다음과 같이 아름다운 기호를 등장시켰습니다.

```
/* 구조체 포인터 */
#include <stdio.h>
struct test {
  int a, b;
};
int main() {
  struct test st;
  struct test *ptr;
  ptr = &st;
  ptr->a = 1;
  ptr->b = 2;
```

```
  printf("st 의 a 멤버 : %d \n", st.a);
  printf("st 의 b 멤버 : %d \n", st.b);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
st 의 a 멤버 : 1
st 의 b 멤버 : 2
```

여기서 새로 등장한 기호는 제가 가장 좋아하는 기호입니다.

```
ptr->a = 1;
ptr->b = 2;
```

위와 같이 `(*ptr).a = 1` 이라는 문장을 `ptr->a = 1` 로 간단히 표현할 수 있습니다. 아래 `ptr->b = 2` 역시 `(*ptr).b = 2` 와 정확히 일치하는 문장입니다. 단순히 사용자의 편의를 위해서 `->` 라는 새로운 기호를 도입한 것 뿐이지요. (이 기호는 위의 우선 순위 표 맨 위에서도 볼 수 있습니다)

자, 그럼 오늘 강좌는 여기서 마치도록 하겠습니다. 사실 이번 강좌는 구조체에 대한 개략 적인 소개만을 전해드린 것이지, 구조체의 진짜 면모는 다음 시간 부터 시작됩니다. 구조체는 정말 잘 쓰면 보물 같은 존재이니 포인터와 더불어 C 언어의 양대 산맥을 이루는 기능이라 말할 수 있습니다. 그럼, 이만 계세요~

생각해보기

문제 1

구조체 안에 또다른 구조체 변수를 설정할 수 있을까요? (난이도 : 中)

문제 2

구조체를 인자로 가지는 함수를 생각해보세요. (난이도 : 中)

구조체를 인자로 받는 함수

안녕하세요 여러분~ 드디어 구조체의 두번째 강의를 시작하게 되었습니다. 지금 강좌를 쓰다가 느낀 건데 제가 구조체를 배웠을 때 에는 정말로 재미있게 배웠던 것 같습니다. 일단 이전 강좌에서도 말했듯이 -> 기호가 상당히 매력적으로 다가왔는데 그것 이외에도 "struct" 라는 단어를 정말 좋아했던 것 같네요. 여러분은 안그러시나요?

구조체 포인터 연습하기

일단, 이번 장의 진도를 나가기 위해선 구조체 포인터에 아주 능숙해 져야 하므로 지난번의 내용을 잠깐 복습하도록 하겠습니다.

```c
/* 포인터 갖고 놀기 */
#include <stdio.h>
struct TEST {
  int c;
};
int main() {
  struct TEST t;
  struct TEST *pt = &t;

  /* pt 가 가리키는 구조체 변수의 c 멤버의 값을 0 으로 한다*/
  (*pt).c = 0;

  printf("t.c : %d \n", t.c);

  /* pt 가 가리키는 구조체 변수의 c 멤버의 값을 1 으로 한다*/
  pt->c = 1;

  printf("t.c : %d \n", t.c);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
t.c : 0
t.c : 1
```

만일 지난번의 강좌를 어렴풋이 나마 기억하고 있는 분들이라면 별로 어려운 내용은 아닐 듯 싶습니다.

```
struct TEST t;
struct TEST *pt = &t;
```

일단 struct TEST 형의 구조체 변수 t 와 struct TEST 형을 가리키는 포인터" pt 를 선언하였습니다. 다시 강조하지만, 우리가 int, char 로 생각하는 것 처럼 struct TEST 도 우리가 창조해 낸 하나의 타입이며, 이를 가리키는 포인터 역시 다른 모든 포인터와 같은 크기라는 것입니다. 즉 pt 는절대로 구조체가 아니며, pt 는 단순히 구조체 변수 t 가 저장되어 있는 메모리 공간의 주소값을 보관하고 있을 뿐입니다.

이 때, pt 는 t 의 주소값을 가지고 있으므로 pt 는 t 를 가리키게 됩니다.

```
(*pt).c = 0;

printf("t.c : %d \n", t.c);
```

이제, pt 가 t 를 가리키고 있으므로 우리는 pt 를 가지고 t 의 값을 마음대로 조작할 수 있게 되었습니다.

이전에 int *pi = &i 를 한 후, *pi 를 쓰면 i 를 간접적으로 나타낼 수 있었듯이 *pt 를 이용하면 pt 가 가리키고 있는 struct TEST 형의 변수, 즉 t 를 나타낼 수 있게 됩니다.

따라서 (*pt).c 를 하면 t 의 멤버 c 를 의미하게 되죠. 이 때, *pt 를 괄호로 감싸주는 이유는 . 이 우선순위가 * 보다 높기 때문에 그냥 *pt.c 라고 쓰면 "pt 의 c 멤버가 가리키는 것" 을 의미하게 됩니다.

아무튼 (*pt).c = 0; 을 통해 우리는 t 의 c 멤버의 값을 성공적으로 바꿀 수 있었습니다.

하지만 (*pt).c 는 너무 쓰기 불편합니다. 항상 *pt 를 괄호로 닫아 주어야 하는데, 괄호는 Shift 를 누르고 키보드의 9 번과 0 번을 눌러야 하니 정말로 손가락도 아프로 불편할 따름이지요. 그래서 훌륭한 C 언어 제작자들은 새로운 편리한 연산자를 만들었습니다.

```
pt->c = 1;

printf("t.c : %d \n", t.c);
```

바로 -> 이죠. -> 연산자의 의미는 "pt 가 가리키는 구조체 변수의 멤버" 를 의미합니다. 따라서 pt->c 는 "pt 가 가리키는 구조체 변수, 즉 t 의 멤버 c" 를 의미하게 됩니다. 따라서 pt->c = 1; 을 통해 우리는 t 의 멤버 c 의 값을 1 로 바꿀 수 있었습니다.

```c
/* 헷갈림 */
#include <stdio.h>
struct TEST {
  int c;
  int *pointer;
};
int main() {
  struct TEST t;
  struct TEST *pt = &t;
  int i = 0;

  /* t 의 멤버 pointer 는 i 를 가리키게 된다*/
  t.pointer = &i;

  /* t 의 멤버 pointer 가 가리키는 변수의 값을 3 으로 만든다*/
  *t.pointer = 3;

  printf("i : %d \n", i);

  /*

  -> 가 * 보다 우선순위가 높으므로 먼저 해석하게 된다.
  즉,
  (pt 가 가리키는 구조체 변수의 pointer 멤버) 가 가리키는 변수의 값을 4 로
  바꾼다. 라는 뜻이다/

  */
  *pt->pointer = 4;

  printf("i : %d \n", i);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
i : 3
i : 4
```

아마 위 예제만 제대로 이해하신 다면 더이상 구조체 포인터 가지고 혼동하는 일은 없을 듯 합니다.
먼저, TEST 구조체의 멤버들 부터 살펴봅시다.

```c
struct TEST {
  int c;
```

```
  int *pointer;
};
```

흠. 쟁쟁한 녀석이 나왔군요. 포인터가 있습니다. 하지만 괜찮습니다. 우리는 포인터를 잘 다루거든요.

```
struct TEST t;
struct TEST *pt = &t;
int i = 0;
```

마찬가지로 pt 는 t 를 가리키게 됩니다.

```
t.pointer = &i;
```

일단, 위 문장에서 t 의 pointer 라는 멤버에는 i 의 주소값이 들어갑니다. 따라서 pointer 는 i 를 가리키게 됩니다. 그렇다면 pointer 를 가지고 i 의 값을 바꾸며 놀 수 있겠죠? 바로 다음 문장을 봅시다.

```
*t.pointer = 3;
```

흠. 우선 순위를 고려하면 . 가 * 보다 높으므로 t.pointer 가 먼저 해석되고 그 다음에 *(t.pointer) 형태로 해석되게 됩니다. 따라서, *t.pointer 를 통해 구조체 변수 t 의 pointer 멤버가 가리키는 변수를 지칭할 수 있게 됩니다.

```
*pt->pointer = 4;
```

. 과 마찬가지로 -> 도 * 보다 우선순위가 높습니다. 즉, *(pt->pointer) 와 *pt->pointer 는 동일한 의미라는 것입니다. 아무튼, pt->pointer 를 통해 "pt 가 가리키는 구조체 변수의 pointer 멤버", 즉 t.pointer 을 의미할 수 *(pt->pointer) = 4 를 통해 pointer 가 가리키는 변수의 값을 4 로 바꿀 수 있게 됩니다.

```
/*

구조체 포인터 연습

 */
#include <stdio.h>
int add_one(int *a);
struct TEST {
  int c;
```

```
};
int main() {
  struct TEST t;
  struct TEST *pt = &t;

  /* pt 가 가리키는 구조체 변수의 c 멤버의 값을 0 으로 한다*/
  pt->c = 0;

  /*
  add_one 함수의 인자에 t 구조체 변수의 멤버 c 의 주소값을
  전달하고 있다.
  */
  add_one(&t.c);

  printf("t.c : %d \n", t.c);

  /*
  add_one 함수의 인자에 pt 가 가리키는 구조체 변수의 멤버 c
  의 주소값을 전달하고 있다.

  */
  add_one(&pt->c);

  printf("t.c : %d \n", t.c);

  return 0;
}
int add_one(int *a) {
  *a += 1;
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
t.c : 1
t.c : 2
```

이제, 마지막으로 구조체 포인터 연습을 해볼깔 합니다.

```
struct TEST t;
struct TEST *pt = &t;
```

이전과 마찬가지로 pt 는 t 를 가리키고 있습니다.

```
add_one(&t.c);
```

그리고 add_one 함수에 t 의 멤버 c 의 주소값을 전달하였습니다. 역시 & 보다 . 이 우선순위가 높으므로 위 식은 &(t.c) 와 동일합니다. 아무튼, add_one 함수에 의해 c 의 값이 1 증가 합니다.

```
add_one(&pt->c);
```

마찬가지로 -> 가 & 보다 우선순위가 높습니다. 따라서, pt 가 가리키는 구조체의 멤버 c 의 값이 1 증가하게 됩니다.

이해가 잘 되시죠?

구조체의 대입

구조체의 복사라 하면 무언가 거창할 것 같지만 사실은 상당히 단순한 내용입니다.

바로, 구조체도 보통의 변수들과 같이 = 를 사용할 수 있다는 것이지요. (= 가 대입 연산자 라는 사실은 기억하시죠??)

```c
#include <stdio.h>
struct TEST {
  int i;
  char c;
};
int main() {
  struct TEST st, st2;

  st.i = 1;
  st.c = 'c';

  st2 = st;

  printf("st2.i : %d \n", st2.i);
  printf("st2.c : %c \n", st2.c);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
st2.i : 1
st2.c : c
```

여러분은 아마도 위 소스 코드를 한눈에 이해하셨을 수 있을 것입니다.

```
struct TEST {
  int i;
  char c;
};
```

멤버가 i 와 c 인 struct TEST 를 정의하였고, 이 구조체의 변수인

```
struct TEST st, st2;
```

st 와 st2 를 정의하였습니다. 그리고 st 의 각 멤버에

```
st.i = 1;
st.c = 'c';
```

를 넣였죠.

```
st2 = st;
```

그리고 우리는 위와 같이 st 를 st2 에 대입하였습니다. 우리가 변수 i 를 j 에 대입하면 i 의 값이 j 에 그대로 복사되듯이, st2 의 멤버 i 의 값은 st 의 멤버 i 의 값과 같아지고, st2 의 멤버 c 의 값은 st 의 멤버 c 의 값과 동일해졌습니다. 이는 상당히 합리적이고 대입 연산자의 역할을 잘 해내는 것 같네요.

```
#include <stdio.h>
char copy_str(char *dest, char *src);
struct TEST {
  int i;
  char str[20];
};
int main() {
  struct TEST a, b;

  b.i = 3;
  copy_str(b.str, "hello, world");
```

```
  a = b;

  printf("a.str : %s \n", a.str);
  printf("a.i : %d \n", a.i);

  return 0;
}
char copy_str(char *dest, char *src) {
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }

  *dest = '\0';

  return 1;
}
```

성공적으로 컴파일 했다면

실행 결과

```
a.str : hello, world
a.i : 3
```

위 코드 역시 구조체의 대입이 무엇인지 잘 이해만 했다면 별 무리 없이 이해하실 수 있으리라 생각합니다.

```
struct TEST {
  int i;
  char str[20];
};
```

위와 같이 `struct TEST` 를 정의하였습니다. 이번에는 `int i` 와 `char str[20]` 을 멤버로 가지고 있습니다.

```
struct TEST a, b;

b.i = 3;
copy_str(b.str, "hello, world");
```

이제 구조체를 정의한 뒤, 위와 같이 각각의 멤버를 초기화 합니다. `copy_str` 함수는 15 - 3 강에서 만들어본 함수죠?

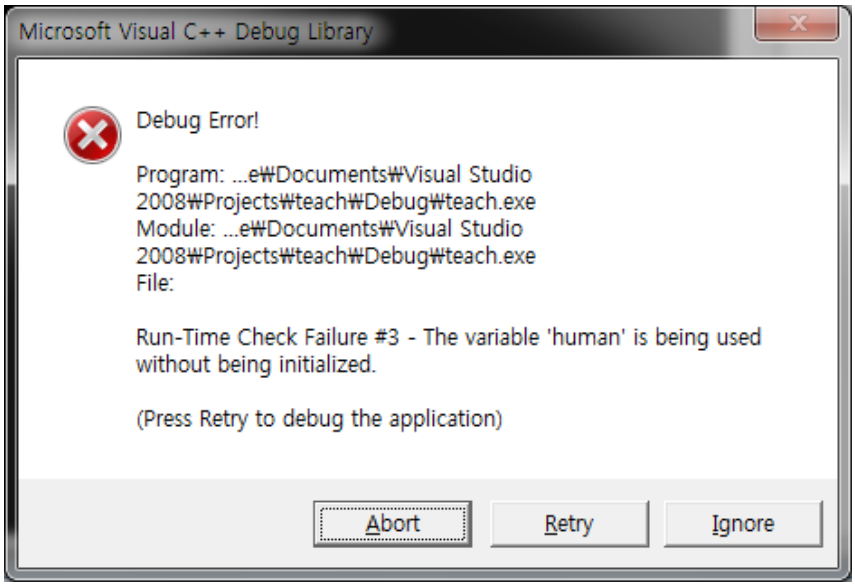
```
a = b;
```

그리고 우리는 위와 같이 b 구조체를 a 구조체에 대입하였습니다. 따라서, b 구조체의 모든 멤버의 데이터가 a 구조체에 일대일로 대응이 되어 값이 복사됩니다. 즉, i 는 i 끼리, str 은 str 의 각 원소 끼리 쭈르륵 복사가 됩니다. 결과적으로 각각의 멤버의 값을 출력해 보면 동일하게 나옵니다.

구조체를 인자로 전달하기

```c
/*구조체를 인자로 전달하기 */
#include <stdio.h>
struct TEST {
  int age;
  int gender;
};
int set_human(struct TEST a, int age, int gender);
int main() {
  struct TEST human;
  set_human(human, 10, 1);
  printf("AGE : %d // Gender : %d ", human.age, human.gender);
  return 0;
}
int set_human(struct TEST a, int age, int gender) {
  a.age = age;
  a.gender = gender;
  return 0;
}
```

성공적으로 컴파일 하였다면 다음과 같은 오류를 만날 수 있으셨을 것입니다.



허걱. 오래간만에 만나는 오류이군요. 오류의 내용을 보자 하니, human 이라는 구조체 변수가 값이 초기화되지 않은 채 사용되었다고 하네요. 일단 소스 부터 살펴보도록 합시다.

```
struct TEST {
  int age;
  int gender;
};
```

일단 우리는 위와 같이 TEST 구조체를 정의하였습니다. 그리고

```
int set_human(struct TEST a, int age, int gender) {
  a.age = age;
  a.gender = gender;

  return 0;
}
```

set_human 이라는 함수를 만들어서 TEST 구조체 변수들을 초기화 하도록 했습니다. 따라서,

```
set_human(human, 10, 1);
```

와 같이 한다면 human 의 age 와 gender 멤버들이 초기화 될 것 처럼보이지요. 하지만 그렇지 않습니다. 왜냐구요? 아마 여태까지 강좌를 잘 따라오셨더라면 한 번에 짐작하실 수 있으실 텐데 말이죠.

바로 제가 13 - 2 강에서 말한 규칙, "특정한 변수의 값을 다른 함수를 통해 바꾸려면 변수의 주소값을 전달해야 한다" 라는 룰을 지키지 않았기 때문입니다. 다시 말해 위 경우에서 a.age = age; 를 했을 때 age 의 값이 바뀌는 것은 실제 main 함수에서의 human 이 아니라 set_human 함수의 a 라는 human 과 별개의 구조체변수의 age 멤버의 값이 바뀌게 되는 것이지요.

따라서 실제 human 구조체변수의 멤버들은 전혀 초기화 되지 않은 채 출력이 실행되어 오류가 발생했습니다.

이를 해결하기 위해서는 역시 human 구조체 변수의 주소값을 인자로 받는 함수를 만들어야 할 것입니다.

```
/* 인자로 제대로 전달하기 */
#include <stdio.h>
struct TEST {
  int age;
  int gender;
};
int set_human(struct TEST *a, int age, int gender);
int main() {
```

```
  struct TEST human;

  set_human(&human, 10, 1);

  printf("AGE : %d // Gender : %d ", human.age, human.gender);
  return 0;
}
int set_human(struct TEST *a, int age, int gender) {
  a->age = age;
  a->gender = gender;

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
AGE : 10 // Gender : 1
```

위와 같이 human 구조체 변수의 멤버의 값들이 제대로 변경되었음을 알 수 있습니다.

```
int set_human(struct TEST *a, int age, int gender) {
  a->age = age;
  a->gender = gender;

  return 0;
}
```

위 set_human 함수는 이전 예제에서의 set_human 함수와는 다르게 구조체의 포인터를 인자로 취하고 있습니다. 그렇기 때문에 set_human 함수를 호출할 때 에서도

```
set_human(&human, 10, 1);
```

위와 같이 human 의 주소값을 인자로 전달하고 있었죠. 따라서, a 는 human 을 가리키게 됩니다. (역시 주의할 점은 a 는 절대로 구조체 변수가 아니라는 것이죠. 단순히 human 구조체 변수가 메모리 상에 위치한 곳의 시작 지점의 주소값을 보관하고 있을 뿐입니다) 아무튼 위와 같이 전달한다면 이제 a-> 를 통해 a 가 가리키고 있는 구조체 변수의 멤버, 즉 위의 경우에서는 human 의 멤버를 지칭할 수 있게 됩니다. 따라서 a->age = age; 를 하게 되면 human 의 age 멤버의 값이 바뀌게 되는 것입니다.

물론 주의할 점은 a->age 와 age 는 다르다는 것이죠. a->age 는 human 구조체 변수의 int 형 멤버 age 를 지칭하는 것이고, age 는 단순히 set_human 함수에서 인자로 받아들여진 int

형의 age 라는 변수를 가리키는 말입니다. 이 둘은 다른 것이고 실제로 컴퓨터 내부에서도 다르게 처리됩니다.

아무튼 위와 같이 제대로 값이 바뀌어서 출력됨을 알 수 있습니다.

```c
/* 살짝 업그레이드*/
#include <stdio.h>
struct TEST {
  int age;
  int gender;
  char name[20];
};
int set_human(struct TEST *a, int age, int gender, const char *name);
char copy_str(char *dest, const char *src);

int main() {
  struct TEST human;

  set_human(&human, 10, 1, "Lee");

  printf("AGE : %d // Gender : %d // Name : %s \n", human.age, human.gender,
         human.name);
  return 0;
}
int set_human(struct TEST *a, int age, int gender, const char *name) {
  a->age = age;
  a->gender = gender;
  copy_str(a->name, name);

  return 0;
}
char copy_str(char *dest, const char *src) {
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }

  *dest = '\0';

  return 1;
}
```

성공적으로 컴파일 했다면

실행 결과

```
AGE : 10 // Gender : 1 // Name : Lee
```

기본적으로 이전의 예제와는 동일하지만 멤버를 하나 더 추가했습니다.

```
struct TEST {
  int age;
  int gender;
  char name[20];
};
```

위와 같이 name[20] 이라는 멤버를 새로 추가해주었습니다.

```
int set_human(struct TEST *a, int age, int gender, const char *name);
```

그리고 set_human 함수에서 name 멤버 역시 같이 초기화해주기 위해 인자로 char * 형의 name 이라는 인자를 추가로 받게 됩니다.

```
set_human(&human, 10, 1, "Lee");
```

이제 TEST 구조체 변수인 human 을 초기화 하기 위해서 set_human 함수를 호출하였습니다.

```
int set_human(struct TEST *a, int age, int gender, const char *name) {
  a->age = age;
  a->gender = gender;
  copy_str(a->name, name);

  return 0;
}
```

위 함수는 a 가 가리키는 구조체 변수의 각 멤버들을 초기화 하게 됩니다. 이 때, main 함수의 human 구초제 변수의 name 멤버를 초기화 하기 위해서는 copy_str 함수를 이용해야 합니다. 이를 위해서는 name 배열의 주소값과, 복사해 넣으려는 문자열의 주소값을 넣어야 하는데 a->name 을 통해 human 구조체 변수의 name 멤버의 주소값과, name (이는 두 번째 인자로 a->name 과 전혀 다른 것이다) 을 통해 복사해 넣으려는 문자열의 주소값을 copy_str 에 전달할 수 있게 됩니다. 아무튼, 위를 통해 성공적으로 human 의 각각의 멤버들을 초기화 할 수 있게 되었죠.

자, 그럼 이번 강좌는 여기서 마치도록 하겠습니다. 아무래도 이번 강좌를 통해 구조체에 대한 확실한 자신감이 생겼으면 하네요. 다음 강좌에서는 구조체에 대해서 조금 더 살펴보고 실습을 해보던지, 아니면 새로운 C 의 기능들에 대해 탐구해 보도록 하죠. (아마 여기까지 도달하신 여러분들은 C 의 모든 고비를 넘겼다고 하셔도 무방합니다. 이제 모든 것이 술술 풀리게 될 것입니다)

생각해보기

문제 1

(이전에 만든)도서 관리 프로그램을만들되,구조체를 이용해 봅시다. 또한 `register_book` 과 같은 함수를 이용하여 책을 등록해봅시다. (난이도 : 下)

문제 2

큰 수를 다루는 구조체를 생각해봅시다. 그 구조체의 이름은 `BigNum` 입니다. `BigNum` 구조체에는 다음과 같은 멤버들이 있을 수 있습니다.

```
struct BigNum {
  int i_digit[100];   // 정수 부분
  int d_digit[100];   // 소수 부분
  int i_total_digit;  // 전체 사용되고 있는 정수부분 자리수
  int d_total_digit;  // 전체 사용되고 있는 소수부분 자리수
  char sign;  // 부호, 0 이면 양수, 1 이면 음수. 0 은 양수로 간주한다.
};
```

- 이 때, `BigNum` 구조체의 변수들의 덧셈, 뺄셈을 수행하는 함수를 작성해보세요 (난이도 : 中上)
- `BigNum` 구조체 변수들의 곱셈을 수행하는 함수를 만들어보세요 (난이도 : 上)
- `BigNum` 구조체 변수들의 나눗셈을 수행하는 함수를 만들어보세요 (난이도 : 最上)

참고로 `BigNum` 구조체를 다룰 때 중요한 점은 수의 크기가 위 배열에 들어가지 않을 정도로 클 때를 적절히 처리해 주어야 한다는 점에 있습니다.

공용체(union)와 열거형(enum)

안녕하세요 여러분! 또 오래간만 입니다. (저 아직 죽지 않고 살아있어요) 구조체를 향한 강좌도 끝을 향해 달려가고 있습니다. 물론 이번 강좌에서는 구조체만을 다루는 것이 아니라 C 언어에서 사용 비중이 그렇게 크지는 않지만 어쨋든 알기는 알아야 하는 기능들에 대해서 배워볼 차례 입니다. 다시말해, 큰 산들은 다 넘었고 이제 우리 앞에는 조그마한 언덕만이 남아 있을 뿐이라는 것이죠 :)

구조체 안의 구조체

```c
/* 구조체 안의 구조체*/
#include <stdio.h>
struct employee {
  int age;
  int salary;
};
struct company {
  struct employee data;
  char name[10];
};
int main() {
  struct company Kim;

  Kim.data.age = 31;
  Kim.data.salary = 3000000;

  printf("Kim's age : %d \n", Kim.data.age);
  printf("Kim's salary : %d$/year \n", Kim.data.salary);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
Kim's age : 31
Kim's salary : 3000000$/year
```

먼저 employee 구조체를 살펴 봅시다.

```c
struct employee {
  int age;
```

```
  int salary;
};
```

위 구조체에는 int 형의 age 와 salary 변수 두 개가 멤버로 되어 있습니다. 다음, company 구조체를 살펴보면

```
struct company {
  struct employee data;
  char name[10];
};
```

와 같이 또다른 구조체 변수를 멤버로 가짐을 볼 수 있습니다. 뭔가 이상하다는 느낌이 들 수 도 있는데, 사실 위와 같이 정의해도 되는 것은 당연한 일입니다. 왜냐하면 제가 이전에도 말했듯이 구조체는 **사용자가 정의한 또다른 형(type)** 이라고 보는 것이기 때문이죠.

구조체 역시 int 나 char 과 같은 하나의 형 입니다. 우리가 만든 형은 이름이 struct employee 라는 것이고, 그 중 data 라는 (구조체) 변수를 생성하였죠. 이는 int a 와 지극히 똑같은 작업입니다.

```
struct company Kim;
```

위와 같이 company 구조체를 정의한 뒤, 'struct company' 형의 변수 Kim 을 정의하였습니다. 이제, Kim 의 멤버들에게 값을 대입해 보아야 겠죠.

```
Kim.data.age = 31;
```

일단 위 문장을 살펴 봅시다. '.' 연산자의 우선 순위는 왼쪽 부터 이므로 Kim.data 가 해석 된 후, (Kim.data).age 가 해석이 됩니다. 다시 말해 "Kim 의 data 멤버의 age 멤버" 로 생각되는 것이지요. 따라서 위와 같이 "Kim 의 data 멤버의 age 멤버" 에 31 의 값을 넣었습니다. 마찬가지로 salary 에 30000 을 넣었습니다.

따라서 위와 같이 출력이 됩니다.

구조체를 리턴하는 함수

구조체는 말그대로 여러분이 창조하신 하나의 타입이기 때문에 int , char 등이 가능했던 모든 것들을 구조체는 그대로 할 수 있습니다. 역시 구조체 형을 리턴하는 함수도 가능하겠지요.

```c
/* 구조체를 리턴하는 함수 */
#include <stdio.h>
struct AA function(int j);
struct AA {
  int i;
};

int main() {
  struct AA a;

  a = function(10);
  printf("a.i : %d \n", a.i);

  return 0;
}

struct AA function(int j) {
  struct AA A;
  A.i = j;

  return A;
}
```

성공적으로 컴파일 했다면

실행 결과

```
a.i : 10
```

먼저 AA 라는 구조체를 정의하였습니다. 편의상 멤버는 int i 로 하나만 가진다고 합시다.

```c
struct AA {
  int i;
};
```

아래는 "struct AA" 형을 리턴하는 함수 function 입니다. 인자로는 int j 를 취합니다.

```c
struct AA function(int j) {
  struct AA A;
  A.i = j;

  return A;
}
```

말그래도 struct AA 형을 리턴하기 때문에 리턴하는 것 역시 struct AA 형의 것이 되야 합니다. 위 함수는 인자로 받는 j 값으로 A 의 i 멤버를 j 의 값으로 초기화 한 후 이를 그대로 리턴합니다.

```
struct AA a;

a = function(10);
```

우리는 main 함수에서 struct AA 타입의 구조체 변수 a 를 정의하였습니다. 그렇다면 a = function(10); 을 통해 function(10) 이 리턴한 구조체의 대입이 일어나게 됩니다. function(10) 은 'i 멤버의 값이 10 인 구조체 변수' 를 리턴하므로 a 의 i 멤버 값은 10 이 됩니다.

구조체 변수의 정의 방법

우리는 여태까지 구조체 변수를 다음과 같이 정의하였습니다.

```
struct Anomynous Var1,
    Var2;  // "struct Anomynous" 형의 변수 Var1 , Var2 를 정의한다.
```

그런데 구조체 변수를 정의하는 방법 중 아래와 같이 색다른 방법을 소개해 드립니다.

```
/*

구조체 변수를 정의하는 색다른 방법.
예제를 이렇게 길게 만든 이유는 소스를 읽으면서 구조체와 조금 더 친해지기 바래서
입니다.  소스를 찬찬히 분석해보세요 ^^

*/
#include <stdio.h>
char copy_str(char *dest, char *src);
int Print_Obj_Status(struct obj OBJ);
struct obj {
  char name[20];
  int x, y;
} Ball;

int main() {
  Ball.x = 3;
  Ball.y = 4;
  copy_str(Ball.name, "RED BALL");

  Print_Obj_Status(Ball);

  return 0;
```

```
}
int Print_Obj_Status(struct obj OBJ) {
  printf("Location of %s \n", OBJ.name);
  printf("( %d , %d ) \n", OBJ.x, OBJ.y);

  return 0;
}
char copy_str(char *dest, char *src) {
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }

  *dest = '\0';

  return 1;
}
```

성공적으로 컴파일 했다면

실행 결과

```
Location of RED BALL
( 3 , 4 )
```

와 같이 나옵니다.

```
struct obj {
  char name[20];
  int x, y;
} Ball;
```

저는 위와 같이 `sturct obj` 라는 구조체를 정의하였고 멤버는 위와 같습니다. 그런데, 맨 아래 `Ball` 은 무엇인가요? 이는 바로 그냥 `sturct obj` 형의 `Ball` 이란 구조체 변수를 정의하라는 뜻입니다. 사실 우리가 `main` 함수 내부에서

```
struct obj Ball;
```

이라고 써왔든 것과 다를 바가 없습니다. 그냥, 위와 같이 구조체 변수를 정의하는 방법도 있다는 것을 알려 드린 것입니다.

나머지 부분은 여러분이 스스로 분석해 보시기 바랍니다.

```c
/* 멤버를 쉽게 초기화 하기*/
#include <stdio.h>
int Print_Status(struct HUMAN human);
struct HUMAN {
  int age;
  int height;
  int weight;
  int gender;
};

int main() {
  struct HUMAN Adam = {31, 182, 75, 0};
  struct HUMAN Eve = {27, 166, 48, 1};

  Print_Status(Adam);
  Print_Status(Eve);
}

int Print_Status(struct HUMAN human) {
  if (human.gender == 0) {
    printf("MALE \n");
  } else {
    printf("FEMALE \n");
  }

  printf("AGE : %d / Height : %d / Weight : %d \n", human.age, human.height,
         human.weight);

  if (human.gender == 0 && human.height >= 180) {
    printf("HE IS A WINNER!! \n");
  } else if (human.gender == 0 && human.height < 180) {
    printf("HE IS A LOSER!! \n");
  }

  printf("------------------------------------------ \n");

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
MALE
AGE : 31 / Height : 182 / Weight : 75
HE IS A WINNER!!
------------------------------------------
```

```
FEMALE
AGE : 27 / Height : 166 / Weight : 48
-----------------------------------------
```

위 예제도 역시 구조체의 잡다한 기능 중 하나를 보여주고 있습니다. 바로 멤버를 초기화 하는 방식인데요, 우리가 이전까지 멤버를 초기화 해온 방법 보다 더 쉽게 할 수 있습니다.

```
struct HUMAN {
  int age;
  int height;
  int weight;
  int gender;
};
```

HUMAN 구조체는 위와 같이 4 개의 int 형 멤버들을 가지고 있습니다.

```
struct HUMAN Adam = {31, 182, 75, 0};
struct HUMAN Eve = {27, 166, 48, 1};
```

그리고 main 내부에서 위와 같이 Adam 과 Eve 를 정의하였죠. 이 때, = {} 를 통해서 중괄호 내부의 정보들이 순차적으로 각 멤버에 대입되게 됩니다. 따라서 Adam 의 경우 age 에는 31 이, height 에는 182 가, weight 에는 75, gender 에는 0 이 들어가게 되죠.

이전 예제에서 배운 초기화 방식에서는 다음과 같이 해주면 됩니다.

```
struct HUMAN {
  int age;
  int height;
  int weight;
  int gender;
} Adam = {31, 182, 75, 0}, Eve = {27, 166, 48, 1};
```

어때요? 간단하지요. 그렇다면 이전 예제의

```
struct obj {
  char name[20];
  int x, y;
};
```

의 경우 어떻게 하면 될까요? 간단 합니다.

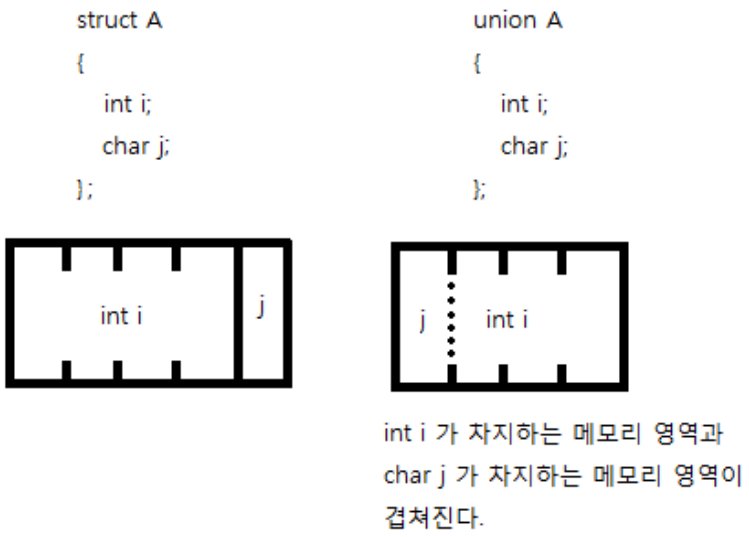
```
struct obj {
  char name[20];
  int x, y;
} Ball = {"abc", 10, 2};
```

와 같이 하게 되면 name 에는 "abc" , x 에는 10, y 에는 2 가 들어가게 됩니다.

여기서 구조체에 관한 이야기는 끝이 납니다. 아. 정말로 길었던 구조체 강좌였습니다. 아마 3 번의 강좌를 걸쳐서 제가 구조체에 관한 모든 지식들을 전달한 것 같습니다.

공용체 (union)

사실 **공용체(union)** 는 그다지 많이 사용하는 기능은 아닙니다만, 그래도 C 언어에서 제공하는 것들 중 하나이니 간단하게 나마 집고 넘어가도록 합시다. 공용체는 구조체와는 달리 메모리를 '공유' 합니다. 이게 도대체 무슨 말인가 하면 아래 그림을 참조해주세요.



위 그림을 보아도 알 수 있듯이 공용체의 각 멤버들의메모리 시작 주소는 모두 동일합니다. 따라서 우리는 위 그림의 union A 의 경우 j 의 값을 변경함으로 써 i 의 값을 변경할 수 있고 마찬가지로 i 의 값을 변경함으로써 j 의 값을 변경할 수 있게 됩니다. 과연 이 말이 진짜 인지 확인해 보도록 합시다.

```
/* 공용체 */
#include <stdio.h>
```

```c
union A {
  int i;
  char j;
};
int main() {
  union A a;
  a.i = 0x12345678;
  printf("%x", a.j);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
78
```

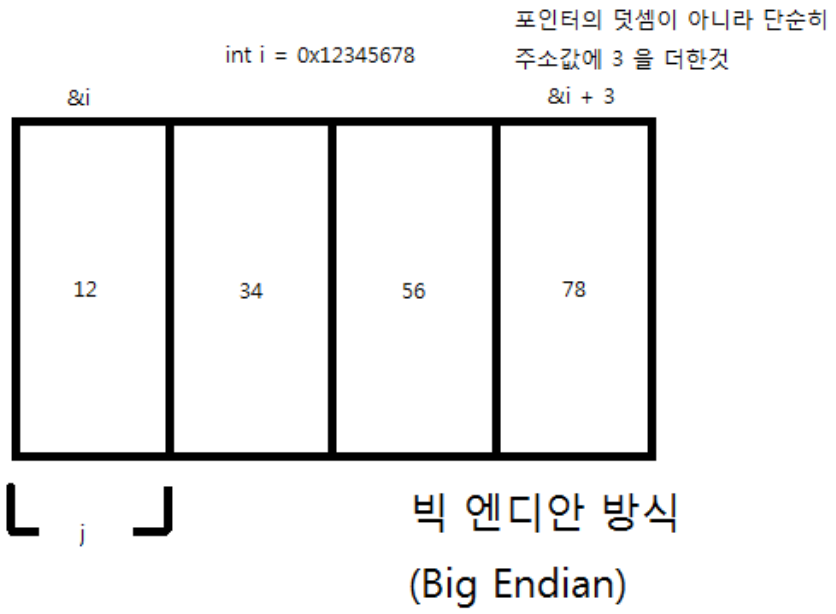
음. 과연 우리는 `a` 의 `j` 멤버 값을 전혀 설정해 주지 않았음에도 불구하고 `i` 에 `0x12345678` 을 대입하자 `j` 의 값이 78 로 잘 나왔습니다. 그런데 이상한 점이 듭니다. 왜 78 이 나왔을 까요? `0x12` 가 나와야 되는 것 아닌가요? 분명히 `i` 와 `j` 에 동일한 주소값에 위치해 있고 `i` 가 `0x12345678` 로 메모리 상에 있다면 `j` 는 처음 두 개인 `0x12` 가 되어야 되는 것 아닌가요? 물론, 여러분의 생각은 옳습니다. 하지만 컴퓨터에서는 수를 이렇게 보관하지 않습니다. 적어도 여러분의 컴퓨터에서는요.

빅 엔디안 (Big Endian), 리틀 엔디안 (Little Endian)

컴퓨터에서 메모리에 수를 저장할 때, 우리가 생각하는 방법, 즉 낮은 주소값에 상위 비트를 적는 방식을 **빅 엔디안** 방식이라고 합니다. 그리고, 우리가 생각하는 방법의 정반대로 높은 주소값에 상위 비트를 적는 방식을 **리틀 엔디안** 이라고 합니다. 현재 대부분은 `x86` 프로세서는 리틀 엔디안 방식을 사용하고 있고 일부 컴퓨터에서만 빅 엔디안 방식을 사용하고 있습니다.[2)]

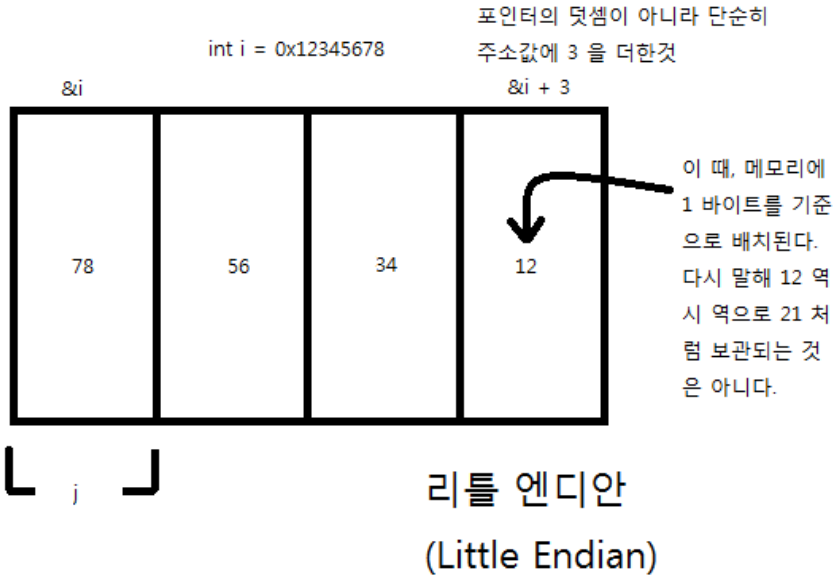
먼저, 빅 엔디안 에서 수를 어떻게 저장하는지 보면!

2) 따라서 여러분의 컴퓨터의 경우 99% 모두 리틀 엔디안을 사용하고 계실 것입니다.



빅 엔디안 에서는 j = 0x12 가 된다.

와 같이 상식적으로 수를 저장하게 됩니다. 하지만 이건 빅 엔디안 방식의 경우이고요, 우리가 대부분 사용하는 프로세서는 리틀 엔디안 방식이므로 리틀 엔디안 방식의 경우를 살펴 보게 되면!



리틀 엔디안 에서는 j = 0x78 이 된다.

와 같이 1 바이트 씩 역으로 보관함을 알 수 있습니다. 따라서 우리가 출력했던 `j` 값은 `0x78` 이 됩니다. 만일 `j` 를 `short` 형을 지정했으면 어떨까요? `0x7856` 이 나올까요? `0x5678` 이 나올까요?

```c
#include <stdio.h>
union A {
  int i;
  short j;
};
int main() {
  union A a;
  a.i = 0x12345678;
  printf("%x", a.j);
  return 0;
}
```

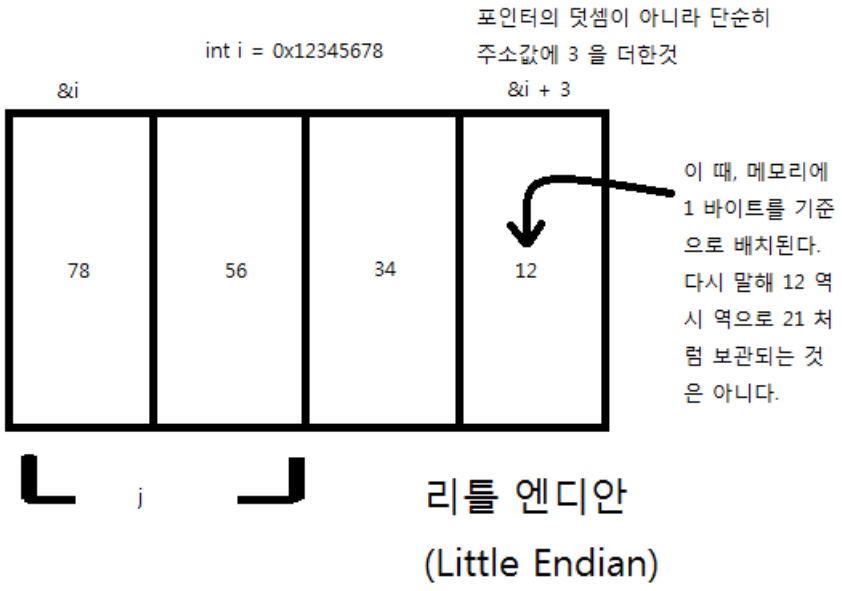
성공적으로 컴파일 했다면

실행 결과

```
5678
```

로 0x5678 이 나오게 됩니다.

이제 다시 머리가 혼란스러워 지기 시작했습니다. 메모리 상에 그대로 살펴 보게 되면 분명히 j 에 해당하는 부분은 0x7856 으로 출력되야 정상이지만 컴퓨터는 '지극히 정상적으로' 0x5678 을 출력하였습니다. 도대체 왜 그럴까요. 그 이유는 간단합니다.



이번에는 j 가 short 형 (2 바이트) 이므로 위와 같이 j 가 2 바이트를 차지하게 됩니다. 이 때 왜 j 의 값을 출력하면 0x7856 이 나오지 않고 0x5678 이 나올까? 이 문제에 대해 고민하고 있을 여러분을 위해 질문을 하나 던지겠습니다. i 의 값을 출력하면 얼마가 나올까요? 당연히 0x12345678

이 될것입니다. 왜냐하면 컴퓨터는 자신이 메모리에 수를 '리틀 엔디안 방식' 을 저장하고 있다는 사실을 알고 있기 때문이죠. 따라서 이를 출력할 때에는 적절한 변환을 취해서 `0x12345678` 을 출력할 것입니다.

`j` 의 경우도 마찬가지 입니다. `j` 는 현재 '78 56' 부분을 가리키고 있지만 컴퓨터는 `j` 가 리틀 엔디안 형식으로 이루어 졌다는 것을 알기 때문에 `j` 를 출력할 때에는 적절히 변환하여 `0x5678` 을 출력하게 될 것입니다.

어때요? 간단 하지요?

공용체에 관한 설명은 여기서 끝납니다. 아마도 공용체를 접할 가능성이 100 번 코딩 하다 보면 1 번 나올까 말까 한데 이를 자세히 짚고 넘어가는 것은 큰 의미가 없다고 생각합니다. 사실 공용체에 대해 배운 것 보다는 엔디안에 대해 배운 것이 훨씬 중요하기 때문에 혹여라도 엔디안에 대해 잊는 일은 없길 바랍니다.

열거형 (Enum)

프로그래밍을 하다 보면 각 데이터에 수를 대응 시키는 경우가 많습니다. 예를 들어 사람을 처리할 때, 남자에는 0, 여자에는 1 을 대응시켜서 처리하거나 색깔을 나타낼 때도 빨강에는 0, 흰색에는 1 등을 대응 시켜서 나타내게 됩니다. 이렇게 수를 대응 시켜서 처리할 때 에는 아래와 같이 헷갈리는 경우가 발생합니다.

```
if (human.gender == 0)  // 사람의 성별이 0 일 때
```

남자에는 0, 여자에는 1 임을 확실하게 기억하고 있다면 상관이 없겠지만 기억하지 못하게 된다면 성에 대해 무엇을 대응 시켰는지 다시 찾아 보아야 된다는 번거로운 일이 발생합니다. 하지만 아래와 같이

```
if (human.gender == MALE)  // 사람의 성별이 남자 일 때
```

와 같이 한다면 확실히 알아 듣기 쉽겠지요. 하지만 문제는 이를 위해 `MALE` 이라는 상수를 설정해야 되고 이 때문에 메모리가 낭비되게 됩니다. 이는 프로그래머의 입장에서 난감한 일이 아닐 수 없지요. C 에서는 이를 **열거형(Enum)**을 도입해서 말끔하게 해결해줍니다.

```
/* 열거형의 도입 */
#include <stdio.h>
enum { RED, BLUE, WHITE, BLACK };
int main() {
  int palette = RED;
  switch (palette) {
    case RED:
```

```
      printf("palette : RED \n");
      break;
    case BLUE:
      printf("palette : BLUE \n");
      break;

    case WHITE:
      printf("palette : WHITE \n");
      break;
    case BLACK:
      printf("palette : BLACK \n");
      break;
  }
}
```

성공적으로 컴파일 했다면

실행 결과

```
palette : RED
```

일단, 열거형을 정의한 부분 부터 살펴 봅시다.

```
enum { RED, BLUE, WHITE, BLACK };
```

열거형을 나타내기 위해서는 enum 을 쓰고 중괄호 안에 각각에 대해 써주면 됩니다. 그렇다면 컴파일러는 열거형에 나타나 있는 각 원소에 0 부터 차례로 정수값을 매겨 주게 됩니다. 즉 RED = 0, BLUE = 1, .. BLACK = 3 와 같이 말이지요. 이제 우리는 이를 자유롭게 이용하면 됩니다.

예를 들어

```
if (palette == 0)  // 현재 파레트의 색이 빨강인지 확인한다.
```

로 했던 것을

```
if (palette == RED)  // 현재 파레트의 색이 빨강인지 확인한다.
```

로 하면 됩니다. 사실 위와 의미는 정확히 똑같지만 프로그래머가 읽을 때 에는 큰 차이가 있게 되죠. 위와 같이 한다고 해서 실질적으로 RED 라는 변수가 메모리에 정해지는 것은 아닙니다. 컴파일 시에 컴파일러는 RED 는 모두 0 로 바꾸고 BLUE 는 모두 1 로 바꾸는 등 변환 작업을 하게 됩니다.

```c
/* 열거형 팁 */
#include <stdio.h>
enum { RED = 3, BLUE, WHITE, BLACK };
int main() {
  int palette = BLACK;
  printf("%d \n", palette);
}
```

성공적으로 컴파일 했다면

실행 결과

```
6
```

열거형에서 처음 수를 0 으로 시작하기 싫다면 어떨까요. 단순히 원하는 수로 해주면 됩니다. 예를 들어 위와 같이

```c
enum { RED = 3, BLUE, WHITE, BLACK };
```

으로 한다면 `RED = 3` 부터 해서 `BLUE = 4, WHITE = 5, BLACK = 6` 이 됩니다. 또한,

```c
enum { RED = 3, BLUE, WHITE = 3, BLACK }
```

으로 한다면 수를 지정한 부분 부터 다시 시작 되는 방식으로 `BLUE = 4, BLACK = 4` 가 됩니다. 참고로 열거형 에서는 언제나 '정수값' 이여야만 합니다.

생각해볼 문제

문제 1

공용체는 도대체 어디에 써먹을 수 있을까요? 이 글을 읽어보세요.

15

변수의 생존 조건과 데이터 세그먼트의 구조

안녕하세요 여러분. 드디어 17 번째 강좌 입니다. 총 20 에서 25 강 까지로 예상하고 있는데 이제 앞으로 얼마 남지 않았군요. 구조체까지 완전히 배웠으니 이제 여러분은 정말로 만들어 볼 것이 많을 것 같네요. 이번 강좌는 단순한 내용이므로 딱히 아주 길지는 않을 것 입니다. 아마도 여태까지 배운 개념들을 환기시키는 정도로 사용될 것 같네요.

변수의 접근 범위

아래의 간단한 소스를 살펴 봅시다.

```c
#include <stdio.h>

void function() {
  int a = 2;
}

int main() {
  int a = 3;
  function();

  printf("a = %d \n", a);
}
```

컴파일 하였다면

실행 결과

```
a = 3
```

와 같이 나옵니다.

```
void function() { int a = 2; }
```

분명히 function 에서 a 에 2 를 대입했는데에도 main 함수에서는 이전의 a 의 값인 3 이 나왔습니다. 이런 결과가 나온 이유는 어떠한 함수 내에서 일반적으로 정의된 변수는 해당 함수 내에서만 접근할 수 있기 때문입니다.

이렇게 해당 **지역** 에서만 접근할 수 있다고 해서 위와 같은 변수들을 지역 변수라고 합니다. 즉 main 함수 안의 a 와 function 안에 a 는 컴파일러가 보기에 다른 변수로 취급됩니다.

```
#include <stdio.h>

int main() {
  int a = 3;
  {
    int a = 4;
    printf("a = %d \n", a);
  }

  printf("a = %d \n", a);
}
```

참고로 중괄호 {} 는 하나의 **지역** 으로 취급되는데, 해당 지역에서 정의된 변수는 바깥 지역에서 정의된 같은 이름의 변수를 가리게 됩니다. 예를 들어서 4 로 초기화된 a 는 바깥의 3 으로 초기화된 a 를 완전히 가리게 됩니다.

따라서 안의 printf 에서는 4 가 출력되고, 바깥의 printf 에는 3 이 출력되겠지요. 위 두 a 는 아예 다른 변수로 취급됩니다. 또한 바깥 지역에서 안쪽 지역에 정의된 변수를 사용할 수 없습니다. 예를 들어서

```
#include <stdio.h>

int main() {
  { int b = 4; }

  printf("b = %d \n", a);
}
```

컴파일 하였다면

컴파일 오류

```
test2.c: In function ‘main’:
test2.c:8:23: error: ‘a’ undeclared (first use in this function)
```

```
    printf("b = %d \n", a);
                        ^
test2.c:8:23: note: each undeclared identifier is reported only
↪  once for each function it appears in
```

위와 같은 컴파일 오류가 발생합니다. 왜냐하면 b 는 printf 가 살고 있는 지역 보다 안쪽에서 정의된 녀석이므로 바깥에서 볼 수 없기 때문이지요!

전역 변수

그렇다면 어떠한 지역에도 속하지 않는 변수를 정의할 수 있을까요? 물론 가능합니다! 그냥 맨 바깥에 정의하면 됩니다.

```c
/* 전역 변수 */
#include <stdio.h>

int global = 0;

int function() {
  global++;
  return 0;
}
int main() {
  global = 10;
  function();
  printf("%d \n", global);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
11
```

어떠한 지역에도 속해있지 않은 변수를 **전역 변수(global variable)** 라 합니다. 전역 변수는 위의 지역 변수와는 달리 코드 어느 곳에서도 접근할 수 있습니다.

먼저 main 에서

```
global = 10;
function();
```

global 의 값을 10 으로 한 후 function 을 호출했습니다. function 에서는 global 의 값을 1 더하는데 따라서 다시 main 에서 global 의 값을 출력 했을 때 에는 11 이 됩니다.

지역 변수의 경우 함수가 종료 될 때 파괴 되었는데, 전역 변수의 경우 프로그램이 시작 할 때 만들어 졌다가 프로그램이 종료 될 때 파괴 됩니다. 전역 변수는 지역 변수와는 달리 메모리의 **데이터 영역(Data segment)**에 할당 됩니다.

한 가지 재미있는 것은 모든 전역 변수들은 정의 시 자동으로 0 으로 초기화 된다는 것입니다.

```c
/* 전역 변수의 초기화 ? */
#include <stdio.h>

int global;
int function() {
  global++;
  return 0;
}
int main() {
  function();
  printf("%d \n", global);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
1
```

위와 같이 1 이 출력됨을 알 수 있습니다. 만일 global 이 지역 변수 같았더라면 일단 컴파일 시에 global 변수가 초기화 되지 않고 사용되었습니다 라는 경고를 냈을 것입니다. 하지만 컴파일러는 전역 변수는 따로 초기화를 하지 않는다면 디폴트로 0 으로 초기화 해버립니다.

따라서 위와 같이 1 이 출력되었죠.

```c
/* 함수 호출 횟수 세기*/
#include <stdio.h>

int how_many_called = 0;
int function() {
  how_many_called++;
  printf("called : %d \n", how_many_called);

  return 0;
}
int main() {
  function();
```

```
  function();
  function();
  function();
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
called : 1
called : 2
called : 3
called : 4
```

위 프로그램에서는 function 이라는 변수가 몇 번 호출 되는지 알려줍니다. how_many_called 라는 변수는 function 함수를 몇 번 이나 호출했는데 카운트 해줍니다. 만일 how_many_called 를 function 함수의 지역 변수로 만들었다면 함수 종료 후 파괴 되므로 정보를 보관할 수 없겠지요.

```
/* 전역 변수의 문제점 */
#include <stdio.h>

int how_many_called = 0;
int how_many_called2 = 0;
int function() {
  how_many_called++;
  printf("function called : %d \n", how_many_called);

  return 0;
}
int function2() {
  how_many_called2++;
  printf("function 2 called : %d \n", how_many_called2);

  return 0;
}
int main() {
  function();
  function2();
  function();
  function2();
  function2();
  function2();
  function();
  function();
  function2();
```

```
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
function called : 1
function 2 called : 1
function called : 2
function 2 called : 2
function 2 called : 3
function 2 called : 4
function called : 3
function called : 4
function 2 called : 5
```

이번에는 또 다른 함수 function2 의 호출 회수도 세는 변수를 지정하였습니다. 전역 변수는 모든 함수에서 접근할 수 있기 때문에 이를 위해 다른 변수 how_many_called2 를 도입하게 되었습니다. 그렇다면 이렇게 10 개의 함수에 대해 각각 호출 회수를 세기 위해서는 아마도 10 개의 전역 변수가 필요하게 됩니다.

이는 심각한 문제가 아닐 수 있습니다. 전역 변수는 모든 함수에서 접근할 수 있기 때문에 전역 변수를 사용할 때에는 매우 주의를 기울여라 합니다. 심지어 위처럼 전역 변수를 수십개 선언 하다 보면 필연적으로 문제가 생기게 마련입니다.

주의 사항

많은 수의 전역 변수를 선언하지 않는 것을 권장합니다.

변수의 생존 기간

앞서 변수가 어떤 범위에서 접근 가능한지 이야기 하였습니다. 그렇다면 이번에는 정의한 변수가 얼마나 살아 있는지 이야기할 차례 입니다.

일반적으로 정의된 변수들은 자신이 정의된 지역을 빠져나갈 때 파괴 됩니다. 이 말이 무슨 말이냐면, 자신이 정의된 위치를 포함하고 있는 {} 를 벗어날 때 해당 변수가 사라지게 된다는 뜻이지요.

예를 들어서 아래 코드를 살펴봅시다.

```c
#include <stdio.h>

int* function() {
  int a = 2;
  return &a;
}

int main() {
  int* pa = function();
  printf("%d \n", *pa);
}
```

성공적으로 컴파일 후에 실행하였다면

실행 결과

```
[1]    30588 segmentation fault (core dumped)  ./test
```

위와 같이 오류가 발생하게 됩니다. 사실 컴파일 시에도 경고 메세지가 나오는데;

컴파일 오류

```
test.c: In function ‘function’:
test.c:5:10: warning: function returns address of local variable
 ↪  [-Wreturn-local-addr]
   return &a;
          ^~
```

위 처럼 함수가 지역 변수의 주소값을 리턴한다고 경고하고 있습니다.

그렇다면 위 코드가 왜 문제인지 살펴보겠습니다.

```c
int* function() {
  int a = 2;
  return &a;
}
```

일단 a 라는 변수는 **지역 변수** 입니다. 따라서 a 가 정의된 지역인 function 을 빠져나가면 a 는 소멸됩니다. 다시 말해 a 를 사용할 수 없다는 뜻입니다.

하지만 아래 처럼 a 의 주소값을 리턴해서 function 외부에서 a 를 사용하려 그런다면 어떨까요?

```
int *pa = function();
printf("%d \n", *pa);
```

이 경우 pa 는 이미 파괴된 변수를 가리키고 있기 때문에 문제가 됩니다. 따라서 위 코드는 프로그램 실행시에 오류를 발생하게 됩니다.

아마도 여러분은 이쯤 부터 **그럼, 지역을 빠져나가도 파괴되지 않는 변수는 없을까** 라는 생각을 하시겠죠? 물론 있습니다. 이를 **정적 변수 (static variable)** 이라 합니다.

정적 변수

```
#include <stdio.h>

int* function() {
  static int a = 2;
  return &a;
}

int main() {
  int* pa = function();
  printf("%d \n", *pa);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
2
```

와 같이 잘 나옵니다.

```
static int a = 2;
```

정적 변수를 선언하기 위해서는 그냥 위와 같이 일반적인 변수 선언 앞에 `static` 만 붙여주면 됩니다. 그리고 해당 변수는 자신이 선언된 범위를 벗어나더라도 **절대로** 파괴되지 않습니다.

```
int *pa = function();
```

따라서 a 가 정의된 지역 밖에서 접근하더라도 a 는 소멸되지 않았기 때문에 2 라는 값이 잘 출력되는 것입니다.

그렇다면 한 가지 궁금한 점은 function 을 여러번 호출하면 a 가 여러번 2 로 초기화 되냐고 물을 수 있습니다. 하지만 a 는 딱 한 번만 초기화 됩니다. 다시 말해 static int a = 2 라는 문장은 **딱 한 번** 실행 되며, 더 놀라운 점은 function 을 실행하지 않더라도 a 라는 정적 변수는 이미 정의되어 있는 상태 입니다.

따라서 이와 같은 정적 변수를 사용하면 아래 처럼 해당 함수가 몇 번 호출되었는지도 쉽게 추적할 수 있습니다.

```c
/* 정적 변수의 활용 */
#include <stdio.h>

int function() {
  static int how_many_called = 0;

  how_many_called++;
  printf("function called : %d \n", how_many_called);

  return 0;
}
int function2() {
  static int how_many_called = 0;

  how_many_called++;
  printf("function 2 called : %d \n", how_many_called);

  return 0;
}
int main() {
  function();
  function2();
  function();
  function2();
  function2();
  function2();
  function();
  function();
  function2();
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
function called : 1
function 2 called : 1
function called : 2
```

```
function 2 called : 2
function 2 called : 3
function 2 called : 4
function called : 3
function called : 4
function 2 called : 5
```

와 같이 동일하게 작동함을 알 수 있습니다.

참고로 정적 변수의 경우 전역 변수 처럼 데이터 영역에 저장되고 프로그램이 종료될 때 파괴됩니다. 또한 전역 변수 처럼 정적 변수도 정의시 특별한 값을 지정해 주지 않는 한 0 으로 자동 초기화 됩니다.

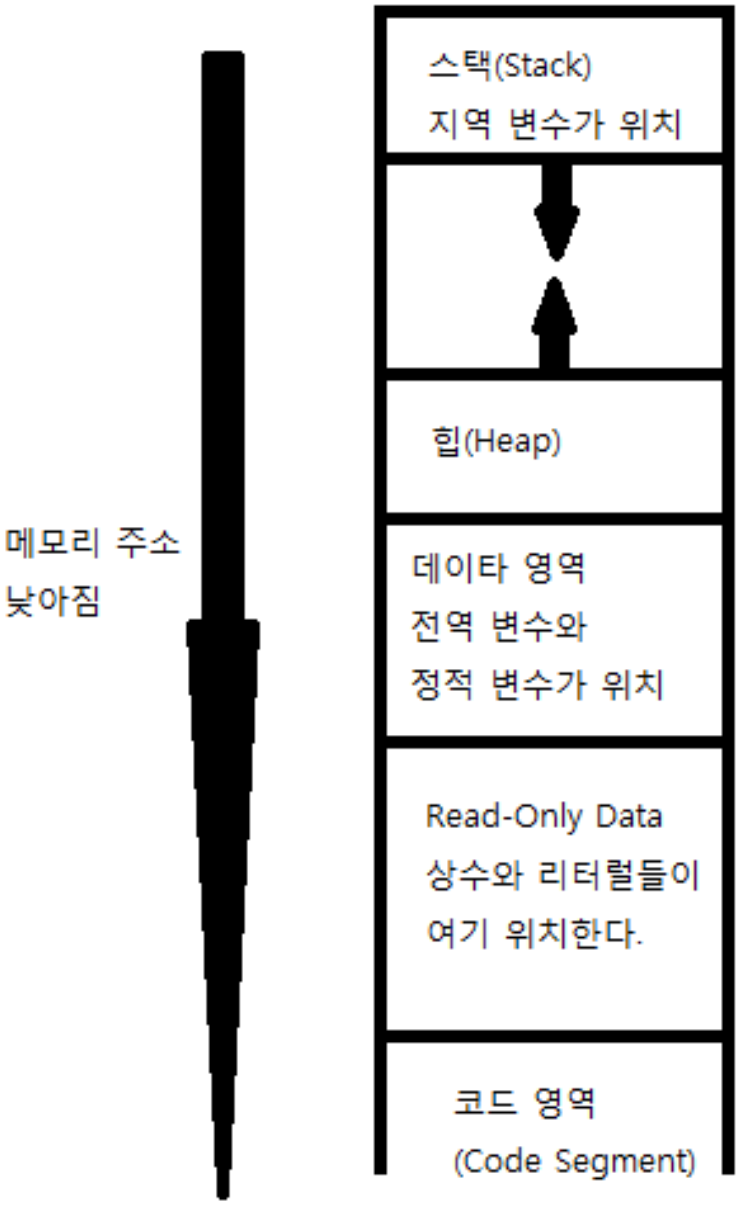
데이터 세그먼트의 구조

주의 사항

아래 내용은 일반적인 운영체제에서 실행 파일이 메모리에 로드될 때 상황을 가정한 그림 입니다. C 언어 자체적으로는 **스택** 이나 **힙** 영역을 따로 구분하지 않습니다. 하지만 대부분의 운영체제에서 프로그램을 실행한다면, 아래 그림 처럼 힙과 스택 영역을 구분해서 만들게됩니다.

프로그램이 실행 될 때 프로그램은 RAM 에 적재 됩니다. 다시 말해 프로그램의 모든 내용이 RAM 위로 올라오게 된다는 것이지요. 여기서 '프로그램의 모든 내용' 이라 하면 프로그램의 코드와 프로그램의 데이터를 모두 의미 하는 것입니다. 이렇게 RAM 위로 올라오는 프로그램의 내용을 크게 나누어서 **코드 세그먼트(Code Segment)** 와 **데이터 세그먼트(Data Segment)** 로 분류할 수 있습니다.

우리가 중점 적으로 살펴볼 것은 데이터 세그먼트입니다. 일단 아래의 그림을 보면



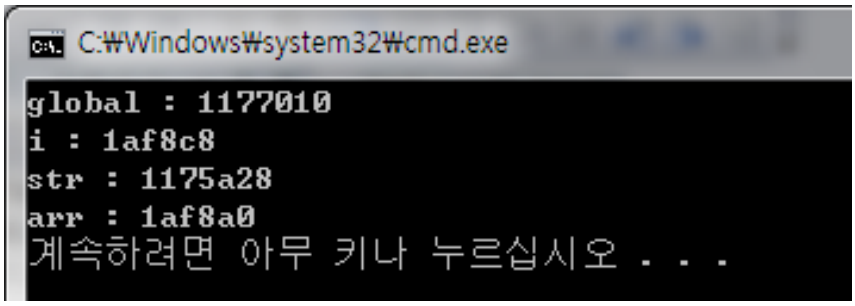
위와 같이 메모리에 배치 되어 있는 것을 알 수 있습니다.

일단 가장 먼저 주목할 부분은 **읽기 전용(Read-Only) Data** 부분 입니다. 이전에 상수와 리터럴에 대해서 이야기 할 때 등장하였는데 이 부분에 저장되는 데이터들은 값이 절대로 변경될 수 없습니다. 다시 말해 궁극적으로 보호 받는 부분 이죠.

그 다음으로 그 위에 전역 변수와 정적 변수가 거처하는 데이터 영역이 있습니다. 그 위에 바로 **힙(Heap)** 이라는 영역이 있는데 이 부분에 대해서는 나중에 설명하도록 합시다. 힙 맨 위를 보면 **스택(Stack)** 이 있습니다. 스택은 지역 변수가 거처하는 곳입니다. 스택의 특징으로는 지역 변수가 늘어나면 크기가 아래로 증가하다가 지역변수가 파괴되면 다시 스택의 크기는 위로 줄어들게 됩니다. 즉, 스택이 늘어나는 방향은 메모리 주소가 낮아지는 방향(아래 방향) 이라 보시면 됩니다.

```c
/* 메모리의 배치 모습 */

#include <stdio.h>
int global = 3;
int main() {
  int i;
  char *str = "Hello, Baby";
  char arr[20] = "WHATTHEHECK";

  printf("global : %p \n", &global);
  printf("i : %p \n", &i);
  printf("str : %p \n", str);
  printf("arr : %p \n", arr);
}
```

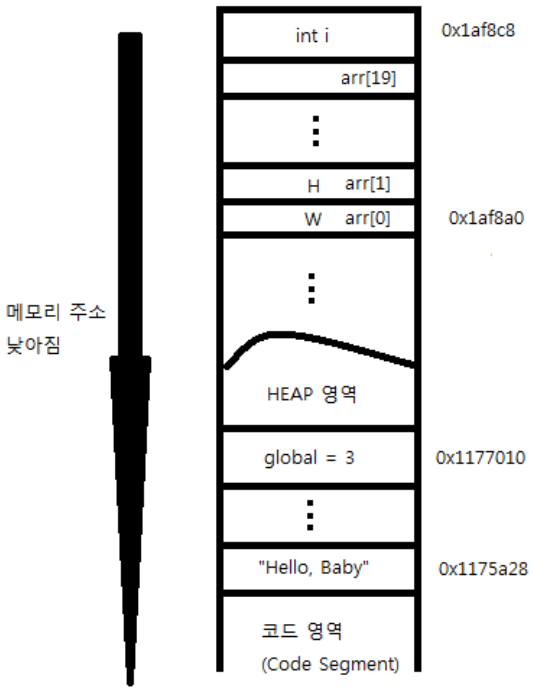
성공적으로 컴파일 했다면



각 변수들의 주소값을 살펴 보면서 과연 메모리에 정말로 그렇게 배치 되었는지 살펴 봅시다. 일단 저의 결과는 여러분의 컴퓨터의 결과와 당연히 차이가 나게 됩니다. 왜냐하면 프로그램 실행 시 그 프로그램이 RAM 어디에 위치하게 될 지는 아무도 모르기 때문이죠. 하지만 그 주소값들만 비교해보도록 합시다.

가장 먼저 읽기 전용(Read Only) 데이터인 `str` 을 봅시다. `str` 에는 "Hello, Baby" 라는 리터럴의 주소값이 들어가 있습니다. 따라서, `str` 의 값을 출력했다면 Read Only 데이터의 위치를 대략 알 수 있겠지요. 여기서는 `0x1175a28` 로 나옵니다. 예상 대로 출력된 주소값들 중 가장 작게 나옵니다. 왜냐하면 `RO data` 는 데이터 세그먼트 맨 아래에 위치해 있기 때문이죠.

두 번째로 전역 변수인 `global` 의 주소값을 살펴보면 `str` 보다는 살짝 크지만 다른 것들 보다는 많이 작다는 것을 알 수 있습니다. 이는 `global` 이 전역 변수로 데이터 영역에 위치해 있기 때문이죠. 세 번째로 `i` 를 보자면 지역 변수 이기 때문에 `stack` 에 존재하고 있습니다. `stack` 의 경우 지역 변수를 추가할 수록 메모리 주소가 작아지는 방향으로 추가가 되므로 `i` 보다 나중에 추가 된 `arr` 의 주소값이 더 작습니다. 이들이 데이터 세그먼트에 배치된 모습을 그림으로 그려 본다면



와 같이 됨을 알 수 있습니다. 어때요? 간단 하지요?

생각해보기

문제 1

스택에 대한 폭넓은 이해를 위해 아래 글을 보는 것을 추천합니다. (동적 할당 부분 전까지만)여기

16

여러 파일로 나누기

안녕하세요 여러분. 저도 강좌를 쓰는 것이 참으로 오래간만 입니다. 현재 저는 강좌를 모두 모두 모아 하나의 pdf 파일로 만드려고 노력중입니다. 이를 위해 LaTeX 를 사용하고 있는데 한국 LaTeX 커뮤니티 (KTUG) 분들께서 훌륭하게 만들어주신 kotex 덕분에 수월하게 파일 제작이 가능합니다. 아무튼 감사하다는 말씀을 드리며 강좌를 시작해 나가보도록 하겠습니다.

현재까지 여러분은 모든 소스 코드를 하나의 소스 파일에서 작성하였습니다. 사실 이는 큰 문제가 아니였습니다. 왜냐하면 우리가 여태 까지 만들었던 프로그램의 총 소스 길이는 그다지 길지 않았고 또 나 혼자 만들기 때문에 하나의 파일에 모조리 작성해도 상관이 없었습니다. 하지만 여러분이 프로그래머가 되어서 회사에서 프로그래밍을 한다면 소스의 길이도 수천~ 수만줄에 이르고, 여러 사람들이 만들기 때문에 파일을 여러개로 나누어야 할 필요성이 있습니다. 물론 파일을 나눌 때 에는 비슷한 작업을 하는 것 끼리 나누는 것이 좋겠죠.

```c
/* 평범한 문장 */
#include <stdio.h>
char compare(char *str1, char *str2);
int main() {
  char str1[20];
  char str2[20];

  scanf("%s", str1);
  scanf("%s", str2);

  if (compare(str1, str2)) {
    printf("%s 와 %s 는 같은 문장 입니다. \n", str1, str2);
  } else {
    printf("%s 와 %s 는 다른 문장 입니다. \n", str1, str2);
  }
  return 0;
}

char compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
```

```
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
hello
hi
hello 와 hi 는 다른 문장 입니다.
```

와 같이 나옵니다.

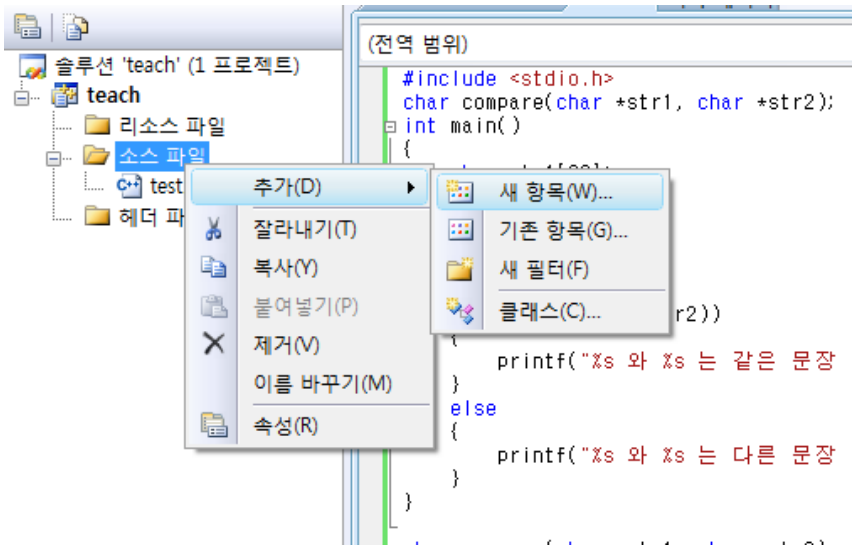
만일 같은 두 문장을 사용하였다면 같다는 메세지가 뜨겠지요. 위 소스 코드는 아주아주 쉬운 내용으로 여태까지 내용을 잘 이수하였다면 잘 이해하실 수 있을 것입니다. 다만,

```
if (compare(str1, str2))
```

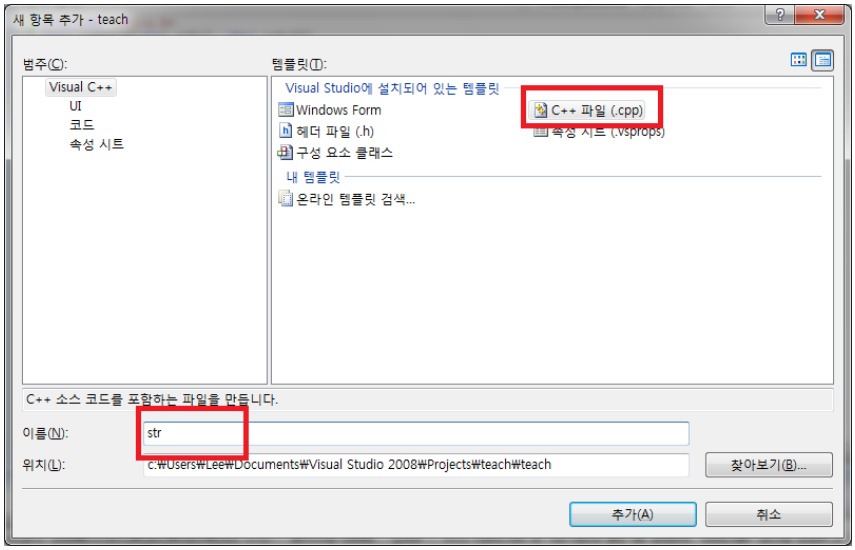
이 부분에서 살짝 갸우뚱 하시는 분들이 있으실 텐데 우리가 만든 compare 함수는 두 문자열이 같으면 1, 다르면 0 을 리턴합니다. 그런데 if 문의 경우 괄호 안의 값이 0 이면 '거짓' 으로, 0 이 아니면 '참' 으로 판단하기 때문에 (이는 이전 강좌에서 이야기 했던 바 입니다. 기억이 안나면 http://itguru.tistory.com/10 를 보세요) 우리가 원하는 결과를 얻을 수 있었던 것이지요.

그렇다면 이번에는 이 강좌의 주제에 맞게 파일을 분할해봅시다. compare 함수는 상당히 다른 일을 하고 있기 때문에 굳이 main 함수와 같은 파일에 둘 필요가 없습니다. 따라서 다른 파일에 만들어 보겠습니다.

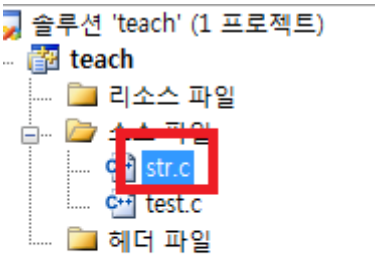
파일 나누기



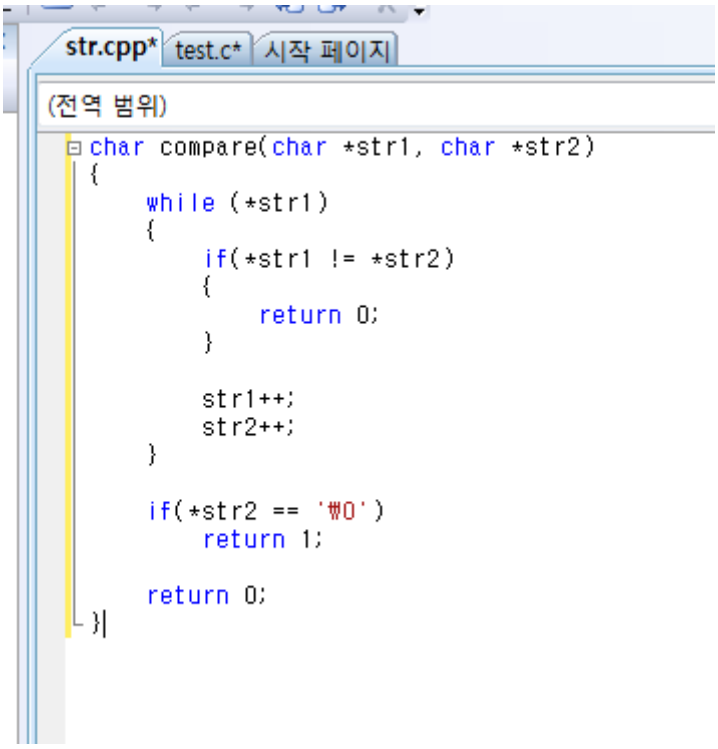
먼저 위의 그림과 같이 오른쪽 파일 목록이 나와 있는 부분에서 '소스 파일' 폴더를 클릭한 후 마우스 오른쪽 클릭을 한 후, '새 항목' 을 위와 같이 누릅니다.



그러면 위와 같이 새 항목을 추가할 수 있는 화면이 뜨는데 우리는 'C++ 파일(.cpp)' 을 선택하고, 이름에 `str` 을 적습니다. `str` 을 적은 이유는 우리가 만든 `compare` 함수는 문자열을 처리하므로 이에 맞게 `str` 이라는 이름을 붙여 주었습니다.



하지만 우리가 하려는 것이 C 언어 이지 C++ 이 아니므로 파일 이름을 `str.c` 로 변경해줍니다. 이는 파일을 클릭한 후 `F2` 를 누르면 파일이름을 변경할 수 있습니다. (또는 마우스 오른쪽 클릭 후 이름 바꾸기를 누른다)



이제, `test.c` (저의 기존 소스를 보관했던 파일 명 입니다. 여러분의 경우 다를 수 있습니다.) 에 쓰여 있던 기존의 `compare` 함수 소스를 잘라옵니다. 즉, `test.c` 에 있던 `compare` 함수 소스는 사라지고 `str.c` 에 `compare` 함수 소스를 넣는 것입니다.

위 작업이 끝나게 된다면 각 파일에는 다음과 같이 소스가 들어가 있을 것입니다.

```
/*
test.c
여러분과 파일 이름은 다를 수 있습니다.
*/
```

```c
#include <stdio.h>
char compare(char *str1, char *str2);
int main() {
  char str1[20];
  char str2[20];

  scanf("%s", str1);
  scanf("%s", str2);

  if (compare(str1, str2)) {
    printf("%s 와 %s 는 같은 문장 입니다. \n", str1, str2);
  } else {
    printf("%s 와 %s 는 다른 문장 입니다. \n", str1, str2);
  }
  return 0;
}
```

```c
/*
str.c
*/

char compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
```

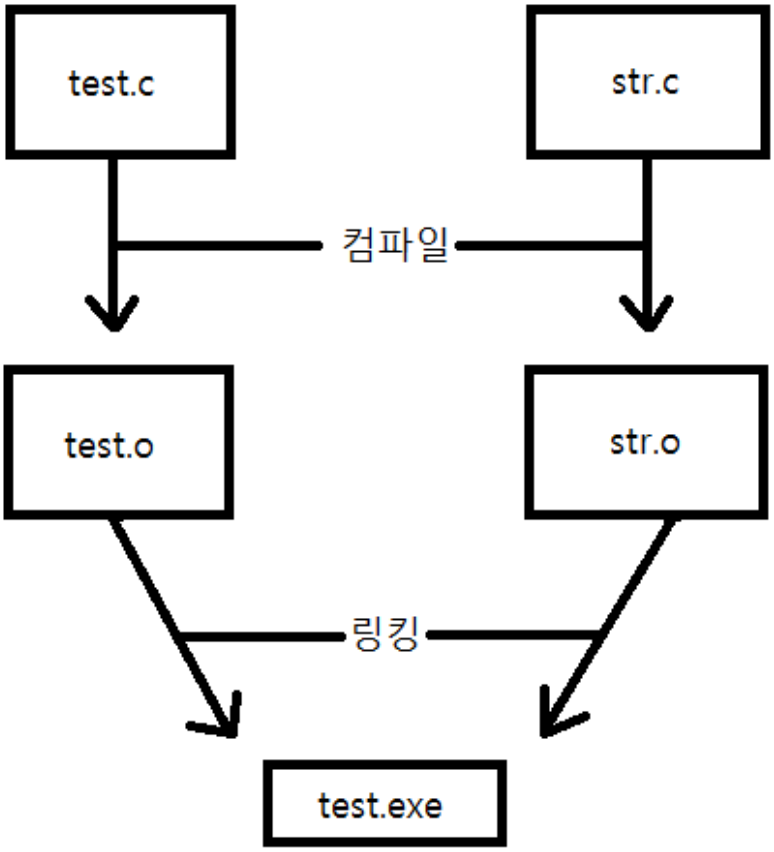
성공적으로 컴파일 하였다면

실행 결과

```
hello
hi
hello 와 hi 는 다른 문장 입니다.
```

아까와 동일한 결과를 얻을 수 있었습니다.

일단 위 프로그램이 어떻게 작동하는지 살펴 봅시다.



아마 예전에 배워서 까먹었을 가능성이 있는데 우리가 실행 파일을 만들기 위해서는 먼저 C 코드를 컴퓨터가 이해할 수 있는 언어로 바꿔주는 **컴파일(compile)** 이라는 과정이 진행됩니다. 이는 단일 소스 코드 전체를 어셈블리어 (기계어와 1 : 1 대응이 되어 있음) 로 변환해 줍니다 (이 때, 목적코드라 불리는 확장자가 `.o` 인 파일이 생성됩니다). 이 과정이 끝나게 되면 **링킹(linking)** 이라는 과정이 진행되는데 말그대로 각기 다른 파일에 위치한 소스 코드들을 한데 엮어서 하나의 실행 파일로 만들어지는 과정이라 생각하시면 됩니다.

링킹 과정에서 특정한 소스 파일에 있는 함수들이 어디어디에 있는지 찾는 과정을 거치게 되는데 예를 들어서 `test.c` 의 경우 `compare` 함수가 어디있는지 찾게 됩니다. (눈치가 빠른 독자라면 `printf` 함수 역시 찾아야 함을 알 수 있는데 이에 대한 설명은 나중에 하겠습니다)

우리의 예제의 경우 `compare` 함수는 `str.c` 에 있기 때문에 **링커**(링킹을 해주는 프로그램) 는 'test.c 에서 `compare` 함수를 호출하는 경우 `str.c` 에서 찾아라' 정도로 처리해 주게 됩니다. 덕분에 우리는 `test.c` 에서 `compare` 함수를 호출하더라도 `str.c` 의 `compare` 함수를 이용할 수 있게 되는 것이지요.

만일 `test.c` 에서

```
char compare(char *str1, char *str2);
```

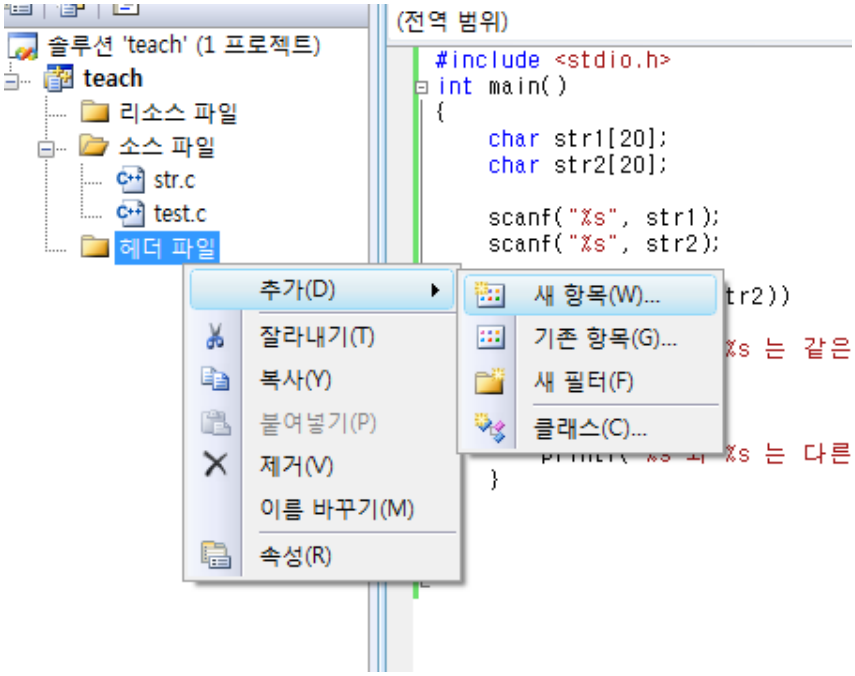
부분을 지워버리면 어떨까요? 이렇게 된다면 컴파일러는 'main 함수에서 compare 함수를 호출하였는데, 도대체 compare 함수는 어떻게 생긴 모양이야!' 가 되어 컴파일 시에 오류가 발생하게 됩니다. 물론 링커의 경우도 compare 함수의 정확한 모양이 무엇인지 알 수 없으므로 오류가 발생하게 되죠.

따라서 이렇게 언제나 함수의 선언을 명시해 주는 것은 매우 중요한 일입니다.

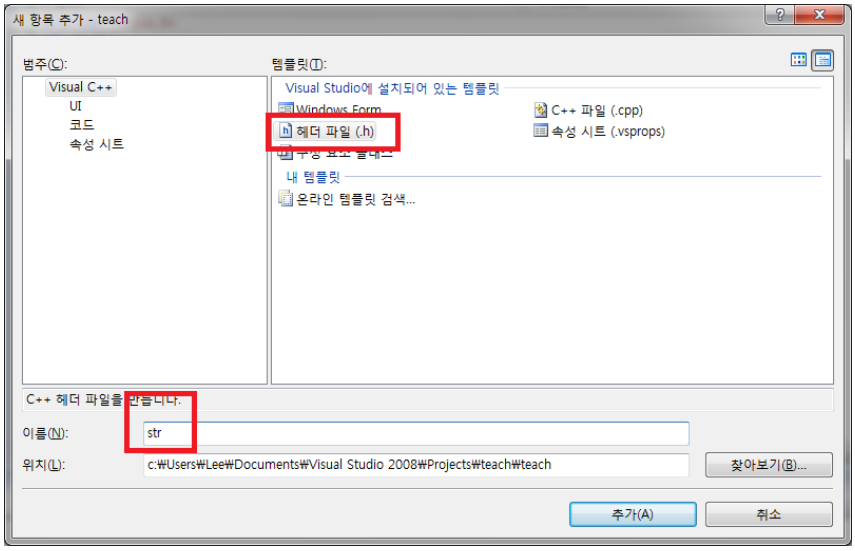
하지만 compare 라는 함수 말고도 copy 라는 함수가 str.c 에 추가되었습니다. 이 함수는 두 문자열을 복사해주는 역할을 합니다. copy 라는 함수를 test.c 에서 이용하기 위해선 역시 copy 함수의 원형을 써주어야 합니다. 이는 상당히 귀찮은 일이지요. 뿐만 아니라 다른 파일에서도 compare 함수와 copy 함수를 이용할 수 있는데 이 파일 역시 이 두 함수의 원형을 써주어야 합니다.

이렇게 귀찮은 작업을 막기 위해 C 에서는 아주 놀라운 해결책을 제시하였는데 바로 **헤더파일 (header file)** 을 이용하는 것입니다. 헤더파일은 다음과 같은 방법으로 만들 수 있습니다.

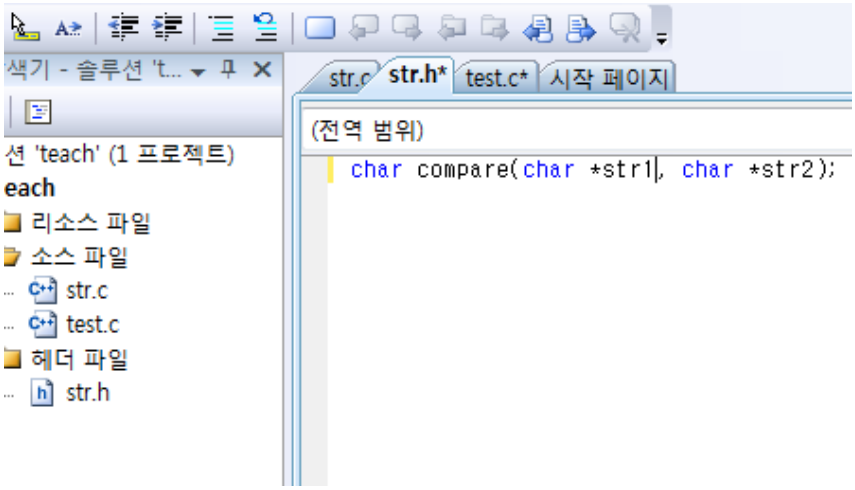
헤더 파일



먼저 기존에 str.c 파일을 추가했을 때 했던 것 처럼 하되 이번에는 헤더파일 폴더에 파일을 추가해보도록 합시다.



헤더파일을 추가하는 것이므로 '헤더파일' 을 선택한 후, 이름에 `str` 을 적습니다.



그리고 헤더파일에는 위와 같이 내용을 기록합니다. 그 후, 각 소스 코드를 다음과 같이 수정해줍니다.

```c
/* test.c */
#include <stdio.h>
#include "str.h"
int main() {
  char str1[20];
  char str2[20];

  scanf("%s", str1);
  scanf("%s", str2);

  if (compare(str1, str2)) {
    printf("%s 와 %s 는 같은 문장 입니다. \n", str1, str2);
  } else {
```

```
    printf("%s 와 %s 는 다른 문장 입니다. \n", str1, str2);
  }
  return 0;
}
```

```
/* str.c */
#include "str.h"

char compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
```

```
/* str.h */
char compare(char *str1, char *str2);
```

성공적으로 컴파일 하였다면

실행 결과

```
hi
hi
hi 와 hi 는 같은 문장 입니다.
```

역시 위와 같이 잘 작동하고 있음을 알 수 있습니다.

먼저 `test.c` 부터 살펴봅시다.

```
#include <stdio.h>
#include "str.h"
```

`#include` 와 같은 명령들은 **전처리기(Preprocessor)** 명령이라고 부르는데 이러한 명령들의 특징은 컴파일 이전에 실행된다는 점 입니다.이 명령은 우리가 지칭하는 파일의 내용을 정확히 100% 복사해서 붙여 넣는다는 점입니다. 따라서 `#include "str.h"` 라는 명령은 `str.h` 에 있었던

내용, 즉 `char compare(char *str1, char *str2);` 로 컴파일이 시작하기 전에 바뀌어 버립니다.

그렇다면 `#include <stdio.h>` 는 어떨까요? 이 역시 똑같습니다. `stdio.h` 에 써있는 내용들이 정확히 복사되어 컴파일 이전에 코드에 붙어버립니다. 그런데 한 가지 이상한 점은 `stdio.h` 는 `<>` 로 감쌌는데, `str.h` 는 왜 `" "` 로 감쌌을까요? 그 이유는 단순한데, `< >` 로 감싸는 헤더파일은 컴파일러에서 기본으로 지원하는 헤더파일의 경우이고 `" "` 로 감싸는 헤더파일은 사용자가 직접 제작한 헤더파일의 경우 입니다.

여러분은 `stdio.h` 에 무엇이 써져 있는지 궁금하지 않으세요? 한 번 제가 그 내용을 올려드리겠습니다. 참고로 여기에서 전체 코드를 보실 수 있습니다.

```
#ifndef _STDIO_H
#define _STDIO_H 1
#define __GLIBC_INTERNAL_STARTING_HEADER_IMPLEMENTATION
#include <bits/libc-header-start.h>
__BEGIN_DECLS
#define __need_size_t
#define __need_NULL
#include <stddef.h>
#define __need___va_list
#include <bits/types.h>
#include <bits/types/FILE.h>
#include <bits/types/__FILE.h>
#include <bits/types/__fpos64_t.h>
#include <bits/types/__fpos_t.h>
#include <bits/types/struct_FILE.h>
#include <stdarg.h>
/* ... 너무 길어서 생략 ... */
```

만일 헤더 파일이라는 것이 존재하지 않았더라면 우리는 `printf` 함수를 이용하기 위해서 위 모든 내용은 아니지만 적어도 `printf` 함수의 원형을 써주어야 하는데 이는

```
_Check_return_opt_ _CRTIMP int __cdecl printf(
  _In_z_ _Printf_format_string_ const char* _Format, ...);
```

로 무지하게 복잡합니다. 아무튼, 이렇게 `printf, scanf` 함수와 같이 매 함수를 쓰기 위해서 위 모든 내용을 쓰는 것 대신에 헤더파일 `include` 하나로 간단하게 해결할 수 있습니다.

보통 헤더파일을 만들 때 에는 그 헤더파일에 있는 함수들이 정의되어 있는 소스 파일의 이름을 따서 짓는 것이 보통입니다. 위 경우 `str.h` 에 선언되어 있는 함수들이 모두 `str.c` 에 정의되어 있으므로 헤더파일의 이름을 `str.h` 로 하였습니다. 또한 한 가지 흥미로운 점은 `str.c` 에서도 `str.h` 를 `include` 하고 있다는 점입니다.

이는 다음과 같은 상황을 방지할 수 있습니다.

/* something.c */ int A() { B(); return 0; } int B() { return 1; }

만일 `something.c` 라는 파일에 위와 같은 소스가 있다고 합시다. 이는 `100%` 오류가 발생됩니다. 왜냐하면 `A()` 함수에서 `B()` 를 호출할 때 `B` 가 무엇인지 뭔지 모르므로 오류가 발생하게 되는 것이지요. (이는 함수 단원에서 공부한 한 바 입니다) 다시 말해 `B()` 를 위에 선언 해주어야 합니다. 아래와 같이요.

/* something.c */ int B(); int A() { B(); return 0; } int B() { return 1; }

하지만 헤더파일을 배웠으니 차라리 이렇게 할 바에는 아래와 같이 하는 것이 훨씬 낫을 것이라는 거죠.

/* something.c */ #include "something.h" int A() { B(); return 0; } int B() { return 1; }

/* something.h*/ int A(); int B();

이와 같은 이유로 `str.c` 에서도 (이 경우 꼭 필요는 없었지만) `str.h` 를 `include` 해준 것입니다.

도서 관리 프로그램 리모델링

그렇다면 이번에는 여태까지 쌓은 지식을 바탕으로 이전에 만들었던 도서 관리 프로그램을 파일을 나누어서 깔끔하게 만들어봅시다.

아래는 기존의 소스 입니다.

```c
#include <stdio.h>
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book);

char compare(char *str1, char *str2);
int borrow_book(int *borrowed);
int return_book(int *borrowed);

int main() {
  int user_choice;        /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */

  /* 각각 책, 저자, 출판사를 저장할 배열 생성. 책의 최대 개수는 100 권*/
  char book_name[100][30], auth_name[100][30], publ_name[100][30];
  /* 빌렸는지 상태를 표시 */
  int borrowed[100];

  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");
```

```
    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);

    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
      add_book(book_name, auth_name, publ_name, borrowed, &num_total_book);
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
      search_book(book_name, auth_name, publ_name, num_total_book);
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
      borrow_book(borrowed);
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
      return_book(borrowed);
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    }
  }

  return 0;
}
/* 책을 추가하는 함수*/
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {
  printf("추가할 책의 제목 : ");
  scanf("%s", book_name[*num_total_book]);

  printf("추가할 책의 저자 : ");
  scanf("%s", auth_name[*num_total_book]);

  printf("추가할 책의 출판사 : ");
  scanf("%s", publ_name[*num_total_book]);

  borrowed[*num_total_book] = 0; /* 빌려지지 않음*/
  printf("추가 완료! \n");
  (*num_total_book)++;

  return 0;
}
/* 책을 검색하는 함수 */
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];

  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
```

```
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);

  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);

  printf("검색 결과 \n");

  if (user_input == 1) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(book_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 2) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(auth_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 3) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < num_total_book; i++) {
      if (compare(publ_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  }
```

```
  return 0;
}
char compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}
int borrow_book(int *borrowed) {
  /* 사용자로 부터 책번호를 받을 변수*/
  int book_num;

  printf("빌릴 책의 번호를 말해주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &book_num);

  if (borrowed[book_num] == 1) {
    printf("이미 대출된 책입니다! \n");
  } else {
    printf("책이 성공적으로 대출되었습니다. \n");
    borrowed[book_num] = 1;
  }

  return 0;
}
int return_book(int *borrowed) {
  /* 반납할 책의 번호 */
  int num_book;

  printf("반납할 책의 번호를 써주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &num_book);

  if (borrowed[num_book] == 0) {
    printf("이미 반납되어 있는 상태입니다\n");
  } else {
    borrowed[num_book] = 0;
    printf("성공적으로 반납되었습니다\n");
  }

  return 0;
}
```

파일들로 나누기 전에 가장 먼저 고려해야 할 사실은 바로 '어떠한 파일들로 나눌 것인가?' 입니다. 파일을 나눌 때 가장 먼저 살펴보아야 할 점은 각 파일들을 특정한 역할을 가지도록 나누는 것인데 우리의 프로그램의 경우 다음과 같이 나누면 될 것 같습니다.main 함수를 가지는 test.c , 도서 관리 함수들을 가지는 book_function.c, 그리고 문자열 관리 함수를 가지는 str.c 로 나누면 될 것 같습니다. 따라서 book_function.h 와 str.h 라는 헤더파일도 가지겠네요.먼저 가장 단순한 str.c 부터 봅시다. str.c 에는 compare 함수가 들어가면 적당할 것 같습니다.

```
/* str.c */
#include "str.h"
char compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }
    str1++;
    str2++;
  }
  if (*str2 == '\0') return 1;
  return 0;
}
```

또한 str.h 에는 compare 함수의 원형이 선언되어 있죠.

```
/* str.h */
char compare(char *str1, char *str2);
```

그럼 이제, 책들을 처리하는 함수들을 모아놓은 book_function.c 를 살펴봅시다. 단순하게 생각해보면 아래와 같이 하면 될 것 같습니다.

```
#include "book_function.h"
/* 책을 추가하는 함수*/
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {
  printf("추가할 책의 제목 : ");
  scanf("%s", book_name[*num_total_book]);
  printf("추가할 책의 저자 : ");
  scanf("%s", auth_name[*num_total_book]);

  printf("추가할 책의 출판사 : ");
  scanf("%s", publ_name[*num_total_book]);
  borrowed[*num_total_book] = 0; /* 빌려지지 않음*/
  printf("추가 완료! \n");
  (*num_total_book)++;
  return 0;
} /* 책을 검색하는 함수 */
int search_book(char (*book_name)[30], char (*auth_name)[30],
```

```
                char (*publ_name)[30], int num_total_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];
  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);
  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);
  printf("검색 결과 \n");
  if (user_input == 1) {
    /*
  i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을          사용자가 입력한
  검색어와 비교하고 있다.
  */ for (i = 0; i < num_total_book; i++) {
      if (compare(book_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  } else if (user_input == 2) {
    /*
   i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을          사용자가
   입력한 검색어와 비교하고 있다.
   */
    for (i = 0; i < num_total_book; i++) {
      if (compare(auth_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }

  } else if (user_input == 3) {
    /*
  i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를          사용자가 입력한
  검색어와 비교하고 있다.
  */
    for (i = 0; i < num_total_book; i++) {
      if (compare(publ_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  }
  return 0;
}
int borrow_book(int *borrowed) {
  /* 사용자로 부터 책번호를 받을 변수*/
  int book_num;
```

```
  printf("빌릴 책의 번호를 말해주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &book_num);
  if (borrowed[book_num] == 1) {
    printf("이미 대출된 책입니다! \n");
  } else {
    printf("책이 성공적으로 대출되었습니다. \n");
    borrowed[book_num] = 1;
  }
  return 0;
}
int return_book(int *borrowed) {
  /* 반납할 책의 번호 */
  int num_book;
  printf("반납할 책의 번호를 써주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &num_book);
  if (borrowed[num_book] == 0) {
    printf("이미 반납되어 있는 상태입니다\n");
  } else {
    borrowed[num_book] = 0;
    printf("성공적으로 반납되었습니다\n");
  }
  return 0;
}
```

그런데 이렇게 하게 되면 100% 오류가 발생하게 됩니다. 왜냐하면 add_book 함수의 경우만 보아도 printf 함수와 scanf 함수를 사용하고 있기 때문이죠. 이 함수들을 사용하기 위해서는 stdio.h 를 꼭 include 해주어야 합니다. 뿐만 아니라 search_book 함수의 경우 compare 함수를 호출하고 있는데 이 때문에 역시 str.h 를 include 해주어야 합니다. 따라서 올바른 book_function.c 의 모습은 아래와 같이 되어야 합니다.

```
/* book_function.c  */
#include <stdio.h>
#include "book_function.h"
#include "str.h"
/* 책을 추가하는 함수*/
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book) {
  printf("추가할 책의 제목 : ");
  scanf("%s", book_name[*num_total_book]);

  printf("추가할 책의 저자 : ");
  scanf("%s", auth_name[*num_total_book]);
  printf("추가할 책의 출판사 : ");
  scanf("%s", publ_name[*num_total_book]);
  borrowed[*num_total_book] = 0; /* 빌려지지 않음*/
  printf("추가 완료! \n");
  (*num_total_book)++;
```

```
  return 0;
} /* 책을 검색하는 함수 */
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];
  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);
  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);
  printf("검색 결과 \n");
  if (user_input == 1) {
    /*
  i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을 사용자가 입력한
  검색어와 비교하고 있다.
  */
    for (i = 0; i < num_total_book; i++) {
      if (compare(book_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  } else if (user_input == 2) {
    /*
  i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을 사용자가
  입력한 검색어와 비교하고 있다.
  */
    for (i = 0; i < num_total_book; i++) {
      if (compare(auth_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  } else if (user_input == 3) {
    /*
  i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를 사용자가 입력한
  검색어와 비교하고 있다.
  */
    for (i = 0; i < num_total_book; i++) {
      if (compare(publ_name[i], user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_name[i], auth_name[i], publ_name[i]);
      }
    }
  }
  return 0;
}
```

```c
int borrow_book(int *borrowed) {
  /* 사용자로 부터 책번호를 받을 변수*/
  int book_num;
  printf("빌릴 책의 번호를 말해주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &book_num);
  if (borrowed[book_num] == 1) {
    printf("이미 대출된 책입니다! \n");
  } else {
    printf("책이 성공적으로 대출되었습니다. \n");
    borrowed[book_num] = 1;
  }
  return 0;
}
int return_book(int *borrowed) {
  /* 반납할 책의 번호 */
  int num_book;
  printf("반납할 책의 번호를 써주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &num_book);
  if (borrowed[num_book] == 0) {
    printf("이미 반납되어 있는 상태입니다\n");
  } else {
    borrowed[num_book] = 0;
    printf("성공적으로 반납되었습니다\n");
  }
  return 0;
}
```

또한 당연하게도 book_function.h 의 모습은 아래와 같겠죠.

```c
/* book_function.h */
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book);
int borrow_book(int *borrowed);
int return_book(int *borrowed);
```

이제 마지막으로 main 함수가 있는 test.c 를 살펴볼 차례 입니다.

main 함수에서는 printf 와 scanf 등과, book_function 에 정의되어 있는 함수들을 사용하고 있고 compare 함수는 사용하지 않으므로 다음과 같이만 해주면 됩니다.

```c
/* test.c */
#include <stdio.h>
#include "book_function.h"
int main() {
```

```
  int user_choice;         /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */
  /* 각각 책, 저자, 출판사를 저장할 배열 생성. 책의 최대 개수는 100 권*/
  char book_name[100][30], auth_name[100][30], publ_name[100][30];
  /* 빌렸는지 상태를 표시 */
  int borrowed[100];
  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");
    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);
    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
      add_book(book_name, auth_name, publ_name, borrowed, &num_total_book);
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
      search_book(book_name, auth_name, publ_name, num_total_book);
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
      borrow_book(borrowed);
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
      return_book(borrowed);
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    }
  }
  return 0;
}
```

어때요? 꽤 간단하지요. 이 작업을 한 후 컴파일 후 실행하면 역시나 잘 작동됨을 알 수 있습니다. 이렇게 여러분은 파일을 나누는 방법에 대해서 알게 되었습니다. 이렇게 파일을 나누어서 처리하게 되면 상당히 체계적으로 프로그래밍을 할 수 있는데 이렇게 하는 프로그래밍을 모듈화 프로그래밍 (modular programming) 이라고 합니다. 즉 프로그램의 각 부분 부분을 나누어서 따로 처리 한다는 의미지요.

우리는 여태까지 헤더 파일에는 함수의 선언 밖에 쓰지 않았지만 사실 헤더파일에도 많은 것들이 올 수 있습니다. 이에 대해서는 다음 시간에 알아보도록 합시다.

생각해 볼 문제

문제 1

위 도서 관리 프로그램을 구조체를 이용하여 만든 것이 있을 것입니다. (http://itguru.tistory.com/60 생각해 볼 문제 참조) 이 역시 위와 같이 파일로 쪼개 보세요.

여러 전처리기 구문과 라이브러리 사용법

안녕하세요 여러분~ 잘 지내셨는지요. 그럼 군말 없이 바로 강의에 들어가겠습니다. 참고로 이게 32 번째 강의인데 이렇게 보니까 참 많이도 썼다고 생각되네요. 30 강 까지 모두 pdf 로 만들었는데 사진을 모두 빼고도 페이지가 380 페이지에 달하내요. 사진을 모두 넣게 되면 적어도 500 페이지 정도는 될 텐데 보기 조금 힘들 것 같네요. 제 블로그에 우수한 성적으로 (?) 방문해 주신 분들께 선물로 나눠드릴라 그랬는데 제본비만 만원이 넘을 것 같아서 그건 좀 힘들겠네요.

헤더 파일

여태까지 헤더 파일에는 오직 함수의 원형들 만을 넣었습니다. 하지만 헤더파일에는 함수의 원형 뿐만이 아니라 아래의 것들도 함께 주로 쓰는 경우가 대다수 입니다. (물론 헤더 파일에도 보통의 C 코드를 집어 넣을 수 있지만 권장하지는 않습니다.)

- 전역 변수
- 구조체, 공용체, 열거형
- 함수의 원형
- 일부 특정한 함수 (인라인 함수.. 나중에 설명함)
- 매크로 (나중에 설명함)

우리는 그 중에서도 위의 3 개 정도만 지금 사용해 보도록 하겠습니다. 다른 것들은 나중에 배워 가면서 익히도록 하죠.

이번에 만들어 볼 것은 Human 이라는 구조체 입니다. Human 구조체에서 가질 정보는, 사람의 이름, 나이, 성별 입니다. 또한, 이 구조체 변수에 대한 정보를 출력하는 함수와, 이 구조체를 설정하는 함수들이 필요합니다.

먼저, Human 구조체 부터 봅시다.

```
/* human.h */
enum { MALE, FEMALE };

struct Human {
  char name[20];
  int age;
  int gender;
```

```
};

struct Human Create_Human(char *name, int age, int gender);
int Print_Human(struct Human *human);
```

human.h 에는 위와 같은 것들이 포함되어 있습니다. 일단 열거형으로 남자와 여자에 대한 정수값들이 선언되어 있으며, Human 구조체, 그리고 한 Human 구조체 변수를 설정하는 Create_Human 함수와 한 Human 에 대한 정보를 출력하는 Print_Human 함수들이 설정되어 있습니다.

그럼 이 함수들에 대한 정보를 가지는 human.c 파일을 봅시다.

```
/* human.c */
#include <stdio.h>
#include "human.h"
#include "str.h"

struct Human Create_Human(char *name, int age, int gender) {
  struct Human human;

  human.age = age;
  human.gender = gender;
  copy_str(human.name, name);

  return human;
}
int Print_Human(struct Human *human) {
  printf("Name : %s \n", human->name);
  printf("Age : %d \n", human->age);
  if (human->gender == MALE) {
    printf("Gender : Male \n");
  } else if (human->gender == FEMALE) {
    printf("Gender : Female \n");
  }

  return 0;
}
```

일단 Human 구조체를 사용하므로 이 구조체에 대한 설명이 들어있는 human.h 와, printf 를 위한 stdio.h, 그리고 copy_str 함수를 위한 str.h 헤더 파일들을 모두 include 해주어야 합니다. 이를 안할 시에 함수를 찾을 수 없다는 오류가 발생하게 됩니다.

str.h 는 단순히 copy_str 함수를 위한 것이므로 str.h 에는 다음과 같이 써있습니다.

```
/* str.h */
char copy_str(char *dest, char *src);
```

또한

```c
/* str.c */
#include "str.h"

char copy_str(char *dest, char *src) {
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }

  *dest = '\0';

  return 1;
}
```

와 같이 함수의 몸체가 나타나 있지요. 자, 그럼 `main` 함수가 위치한 `test.c` 를 봅시다.

```c
#include <stdio.h>
#include "human.h"
int main() {
  struct Human Lee = Create_Human("Lee", 40, MALE);

  Print_Human(&Lee);

  return 0;
}
```

상당히 간단합니다. 이는 우리가 파일을 잘 나누었기 때문입니다. 좋은 프로그램일 수 록 `main` 함수에서 하는 일이 적어집니다.

성공적으로 컴파일 하였다면

실행 결과

```
Name : Lee
Age : 40
Gender : Male
```

위에서 소스를 설명하면서 다 말했기 때문에 굳이 설명할 부분은 없습니다만, 만일 위 소스에서 하나라도 이해가 되지 않는 부분이 있다면 반드시 이전 강의를 복습해주시기 바랍니다.

이제 파일을 분할하는 과정에 대해 배웠으니 파일을 분할하는 것을 습관을 들이시기 바랍니다. 처음에 함수를 배웠을 때 프로그래밍이 상당히 편해진 것 처럼 파일을 분할하게 되면 프로그래밍이 상당히 편해지는 것을 느끼실 수 있을 것입니다. 각 소스 파일에 정확히 무엇을 하는지 나타내주는 것이 중요합니다.

다른 사람이 만들어 놓은 것 쓰기

이번에는 파일을 분할하는 것 만큼이나 중요한 것에 대해 알아볼 시간 입니다. 바로 '다른 사람이 만들어 놓은 함수들' 을 사용하는 방법에 대해서 말이지요. 이렇게 다른 사람들이 만들어 놓은 것을 가리켜서 **라이브러리** 라고 합니다. 우리가 도서관에 가서 책을 고르듯이, C 프로그래밍에서 우리는 원하는 함수를 라이브러리에서 찾아 사용할 수 있습니다. 이는 정말로 편리한 일이지요. 시시 콜콜하게 함수들을 귀찮게 만들 필요가 없다는 말입니다.

아래 예제는 기존에 우리가 `copy_str` 을 이용하여 `str1` 에 `str2` 를 복사하는 과정을 나타냈습니다.

```c
/* test.c */
#include <stdio.h>
#include "str.h"
int main() {
  char str1[20] = {"hi"};
  char str2[20] = {"hello every1"};

  copy_str(str1, str2);

  printf("str1 : %s \n", str1);

  return 0;
}
```

```c
/* str.h */
char copy_str(char *dest, char *src);
```

```c
/* str.c */
#include "str.h"
char copy_str(char *dest, char *src) {
  while (*src) {
    *dest = *src;
    src++;
    dest++;
  }

  *dest = '\0';

  return 1;
}
```

성공적으로 컴파일 했다면

실행 결과

```
str1 : hello every1
```

일단 위와 같이 잘 복사되었음을 알 수 있습니다. 하지만 이는 정말로 귀찮은 일이 아닐 수 없습니다. 문자열을 복사하는 과정은 정말로 많이 쓰이는 것입니다. 이렇게 문자열 복사가 필요할 때 마다 copy_str 함수를 만들어서 쓴다면 참으로 귀찮은 일이 아닐 수 없습니다. 하지만 정말 다행스럽게도, 사람들은 이 역할을 하는 함수를 '미리' 만들어 놓았습니다.

```c
/* 라이브러리의 사용 */
#include <stdio.h>
#include <string.h>
int main() {
  char str1[20] = {"hi"};
  char str2[20] = {"hello every1"};

  strcpy(str1, str2);

  printf("str1 : %s \n", str1);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
str1 : hello every1
```

와 같이 위와 똑같이 나옵니다.

```c
#include <string.h>
```

위 명령은 string.h 파일에 있는 내용을 모두 가져다 붙인다 라는 의미를 가지고 있습니다. 그런데 이 string.h 파일에는 '문자열을 처리하는데 관련된 함수들의 원형' 모음이 있습니다. 따라서 우리는 이 파일을 include 시킴으로써 문자열을 처리하는 여러가지 편리한 함수들을 사용할 수 있게 됩니다. 우리가 str.h 를 include 해서 copy_str 을 사용할 수 있었던 것과 일맥 상통합니다. 우리는 여기서 strcpy 라는 함수를 사용했습니다.

```c
strcpy(str1, str2);
```

이 함수는 copy_str 과 사용법이 정확히 똑같습니다. 문자열을 복사하고자 하는 곳의 주소값을 첫번째 인자로, 복사가 되는 문자열의 주소값을 두번째 인자로 주면 됩니다.

이렇게 사람들이 미리 만든 함수들의 모임을 가리켜서 '라이브러리' 라고 합니다. 우리가 현재 사용한 라이브러리는 **문자열(string) 라이브러리** 입니다. 그렇다면 stdio.h 도 라이브러리 일까요? 맞습니다. 이는 **입출력 라이브러리** 로 입력과 출력에 관련된 함수들을 모아놓았습니다. 대표적으로 prinf 와 scanf 가 있지만 이전에 잠깐 소개했던 getchar() 함수나 puts() 등등 수 많은 함수가 여기에 정의되어 있습니다. 이 목록은 제 블로그 C 언어 레퍼런스를 참조하세요.

```c
/* strcmp 함수 */
#include <stdio.h>
#include <string.h>
int main() {
  char str1[20] = {"hi"};
  char str2[20] = {"hello every1"};
  char str3[20] = {"hi"};

  if (!strcmp(str1, str2)) {
    printf("%s and %s is equal \n", str1, str2);
  } else {
    printf("%s and %s is NOT equal \n", str1, str2);
  }

  if (!strcmp(str1, str3)) {
    printf("%s and %s is equal \n", str1, str3);
  } else {
    printf("%s and %s is NOTequal \n", str1, str3);
  }

  return 0;
}
```

성공적으로 살펴 보았다면

실행 결과

```
hi and hello every1 is NOT equal
hi and hi is equal
```

이번에 사용해본 함수는 strcmp 함수 입니다.

이 함수는 두 문자열을 비교해서 두 문자열이 같다면 0 을 다르면 0 이 아닌 값을 리턴하게 되어 있습니다.이 함수의 사용법도 이전에 우리가 만들었던 compare_str 함수와 동일합니다. 첫 번째와 두 번째 인자에는 비교할 문자열들의 주소를 넣어주면 됩니다.

이렇게, 다른 라이브러리의 함수들을 사용하니 상당히 편리합니다. 이번에는 string 라이브러리 말고도 다른 여러가지 라이브러리들이 많은데 이 들에 대한 정보는 여러분이 직접 찾아보세요

친구들

여태까지 우리는 #include 라는 명령에 대해 알아보았습니다. 이렇게 # 이 들어간 명령들은 '전처리기 명령' 이라고 하는데 전처리기의 의미는 컴파일 이전에 처리된다는 뜻 입니다. 즉, 컴파일이 되기 이전에 #include 라는 부분은 #include 에 해당하는 파일의 소스 코드로 정확히 바뀝니다.

이전 강좌에서 stdio.h 의 내용을 썼었을 때, #include 말고도 # 이 들어가 있는 엄청나게 많은 수의 명령들을 볼 수 있었습니다. 예를 들면 #define, #ifdef 등등 이죠. 이번에는 이러한 다양한 종류의 전처리기 명령들에 대해 알아보도록 합시다.

#define

```c
/* #define */
#include <stdio.h>
#define VAR 10
int main() {
  char arr[VAR] = {"hi"};
  printf("%s\n", arr);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
hi
```

여러분은 아마도 배열을 정의하는데 대괄호 안어에 수가 아닌 값이 들어왔는데도 컴파일이 어떻게 잘되고 실형 역시 잘되었는지 놀랐었을 수 도 있습니다. 배열을 정의할 때 대괄호 안에는 언제가 수가 와야 합니다. 심지어 상수 조차 올 수 없습니다. 하지만 위 경우 어떻게 된 것일까요?

#define 명령은 다음과 같이 사용합니다.

```
#define 매크로이름 값 // 전처리기 문들은 끝에 ; 를 붙이지 않습니다!!
```

이는 소스 코드에서 '매크로이름' 에 해당하는 부분을 '값' 으로 대체하게 되는 것입니다. 물론, 전처리기 명령이기 때문에 컴파일 이전에 정확하게 대체됩니다. 따라서,

```c
#include <stdio.h>
#define VAR 10
int main() {
  char arr[VAR] = {"hi"};
  printf("%s\n", arr);
  return 0;
}
```

라는 문장은

```c
#include <stdio.h>
int main() {
  char arr[10] = {"hi"};
  printf("%s\n", arr);
  return 0;
}
```

과 완전히 똑같은 문장 입니다. 이 작업이 컴파일 이전에 처리되기 때문에 컴파일러 입장에서는 `arr[10]` 이라는 문장을 처리하는 것과 똑같으므로 오류 없이 정확하게 수행될 수 있다는 것이죠.

#ifdef, endif

`ifdef` 와 `endif` 는 무언가 `if` 문과 관련이 있을 것 같습니다. `if` 문 처럼 특정한 조건에만 수행이 되겠지요.

```c
/* ifdef */
#include <stdio.h>
#define A
int main() {
#ifdef A
printf("AAAA \n");
#endif
#ifdef B
printf("BBBB \n");
#endif
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
AAAA
```

만일, #define 부분을 #define A 에서 #define B 로 바꿔보면

실행 결과

```
BBBB
```

와 같이 나옵니다.상당히 재미있지요. 일단 ifdef 는 다음과 같은 형식으로 사용됩니다.

```
#ifdef /* 매크로 이름 */
/* (매크로 이름)이 정의되었다면 이 부분이 코드에 포함되고 그렇지 않다면 코드에
 * 포함되지 않는다. */
#endif
```

언제나 ifdef 는 endif 와 짝을 지어서 사용하는데, ifdef 에서 지정한 매크로가 정의되어 있다면 ifdef 와 endif 속에 있는 코드가 포함되고 그렇지 않다면 코드에 포함되지 않는 것으로 간주됩니다. #define A 를 통해 A 가 정의 되어 있다면

```
#ifdef A
printf("AAAA \n");
#endif
```

부분은 전처리기에 의해

```
printf("AAAA \n");
```

로 바뀌지만

```
#ifdef B
printf("BBBB \n");
#endif
```

부분은 소스에 포함되어 있지 않은 것으로 간주되어 컴파일러 입장에서는 마치 주석 처럼 무시됩니다.만일 #define A 대신에 #define B 를 하게 된다면 반대의 상황이 연출되고, 둘 다 define 해주게 된다면 둘 다 코드에 포함이 되겠지요. 이와 같은 기능을 도대체 왜 만들었냐고도 물어볼 수 있는데 사실 이 '조건부 컴파일 (특정한 조건에 따라 컴파일 되는 부분이 다른 것)' 은 상당히 유용하게 쓰일 수 있습니다.

예를 들어 계산기 프로그램을 만드는데, 계산기 모델 마다 조금씩 메모리와 CPU가 틀려서 어떤 계산기에는 `double` 을 사용할 수 있지만 어떤 모델에서는 `float` 밖에 사용할 수 없다고 합시다.

그렇다면 각각 이 계산기를 위해 다음과 같이 소스를 짜야 할 것입니다.

```
/*
계산기 모델 1 을 위한 코드
calculator1.c
*/
float var1, var2;
// do something
```

```
/*
계산기 모델 2 을 위한 코드
calculator2.c
*/
double var1, var2;
// do something
```

하지만 조건부 컴파일을 이용하면 이 두 개의 파일로 나누어서 해야 했던 작업을 다음과 같이 줄일 수 있습니다.

```
#define CACULATOR_MODEL_1

#ifdef CALCULATOR_MODEL_1
float var1, var2;
#endif
#ifdef CALCULATOR_MODEL_2
double var1, var2;
#endif;
// do something
```

이 때, `define` 되는 것이 무엇이냐에 따라 간단히 무엇을 컴파일 할 것인지를 나타낼 수 있습니다.사실 `ifdef` 와 `endif` 가 사용되는 경우는 이것보다 훨씬 많지만 일단 여기서 매듭 짓기로 하겠습니다.

위 조건부 컴파일에서 #else 라는 것도 사용할 수 있는데 이는 #ifdef 의 경우 이외의 나머지 것들을 처리하는 것입니다. 이 역시 #endif 로 항상 끝을 맺어주어야 합니다. 예를 들면 아래와 같지요.

```
#ifdef CALC_1
// do something
#else
// do something 'else'
#endif
```

또한 `#ifdef` 의 친구로 `#ifndef` 도 있는데 이는 '매크로가 정의되어 있지 않다면' 참이 됩니다. `#ifdef` 의 정 반대이지요.이 기능들에 대해서는 나중에 좀더 큰 프로젝트를 진행하면서 차근 차근 알아가보도록 합시다.

생각해보기

문제 1

헤더 파일이 두 번 중복되서 `include` 되지 않기 위해서는 헤더파일에 어떠한 조건문을 넣으면 좋을지 생각해보세요.

17

void 타입과 main 함수에 대한 이해

안녕하세요 여러분. 그동안 잘 지내셨는지요? 책 만드는 것도 어느 정도 진척이 되었고 Latex 도 어느정도 능숙하게 다룰 줄 알아서 꽤 괜찮게 만들 수 는 있었는데 표지가 문제네요. 혹시 '씹어먹는 C 언어' 를 위한 멋진 표지를 만드실 분을 찾고 있으니 혹시 좋은 아이디어가 있으신 분들은 kev0960@gmail.com 으로 꼭 메일을 보내주시기 바랍니다.

리턴값이 없는 함수

때로는 우리가 만드는 함수가 리턴값이 없을 수 도 있습니다. 예를 들어서 어떠한 int 변수에 1 을 더하는 함수를 생각해봅시다.

```
#include <stdio.h>

void add_one(int* p) {
  (*p) += 1;
}

int main() {
  int a = 1;
  printf("Before : %d \n", a);
  add_one(&a);
  printf("After : %d \n", a);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Before : 1
After : 2
```

와 같이 나옵니다. 위 add_one 함수는 인자로 전달된 포인터가 가리키는 것의 값을 1 증가 시킨 뒤 종료됩니다. 그리고 return 을 수행하는 문장이 없습니다. 왜냐하면

```
void add_one(int* p) {
```

위와 같이 void 형으로 선언되어 있기 때문이죠. 영어 사전을 찾아보면 void 는 진공, 공허라는 뜻을 의미합니다. C 언어에 적용해서 생각해보자면 무(無)의 타입을 의미한다고 보실 수 도 있지요. 만일 이 함수가 무언가 리턴을 한다면 그 무언가는 타입이 있어야 되기 때문에 이 void 함수가 가능한 형태는 **아무것도 리턴하지 않는 함수** 가 됩니다.

따라서 다음과 같은 문장은 모두 틀린 셈입니다.

```
void a();
int main() {
  int i;
  i = a();

  return 0;
}

void a() {}
```

즉 함수 a 가 리턴하는 값이 없으므로 i 의 값에 a 의 리턴값을 대입할 수 없습니다. 이 모두 오류로 처리됩니다.

void 형 함수는 많은 곳에서 사용 됩니다. 주로, **리턴을 할 필요가 없는 함수** 들의 경우가 대부분이죠. 예를 들어서 두 변수의 값을 교환하는 함수를 생각해봅시다. 아마 여러분은 여태까지 다음과 같이 함수를 만들었을 것입니다.

```
int swap(int *a, int *b) {
  int temp;

  temp = *a;
  *a = *b;
  *b = temp;

  return 0;
}
```

하지만 swap 함수는 리턴할 필요가 전혀 없죠. 단순히 두 수의 값만 바꾸면 끝인데 뭐하러 귀찮게 리턴을 하냐 말이죠. 오히려 불필요하게 swap 의 리턴 타입이 있다면 swap 함수를 사용하는 사람 입장에서 이 함수의 리턴값은 무슨 의미지? 를 생각해야 합니다.

```
void swap(int *a, int *b) {
  int temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```

하지만 위와 같이 void 로 함수를 만든다면 그런 걱정은 깔끔하게 날려버릴 수 있습니다. 따라서 이렇게 리턴값이 필요 없는 곳에서 void 함수를 이용하는 것이 좋습니다.

void 형 변수

```
/* void 형 변수?? */
#include <stdio.h>
int main() {
  void a;

  a = 3;

  return 0;
}
```

성공적으로 컴파일 하였다면

컴파일 오류

error C2182: 'a' : 'void' 형식을 잘못 사용했습니다.

와 같은 오류 메세지를 보게 됩니다. 우리가 위에서 void 형 함수에 대해 살펴 보았습니다. 그렇다면 void 형의 변수도 정의할 수 있을 것 같은데 사실 이는 오류 입니다. 컴파일러가

```
int a;
```

라는 문장을 보게 된다면 컴파일러는 '아, int 형의 변수 a 를 선언하는 구나. 메모리 상에 미리 4 바이트의 공간을 마련해 놓아야지' 라고 생각할 것입니다. 그런데

```
void a;
```

를 보게 된다면, '응? 이 변수의 타입은 뭐지?' 라고 생각하게 되죠. 다시 말해 이 변수를 위해서 메모리 상에 얼마나 많은 공간을 설정해 놓아야 하는지 모르게 되는 셈입니다. (참고로 컴파일 때 모든 변수들의 메모리 상의 위치가 결정 되어야 합니다) 따라서 이와 같은 형식은 틀리게 된 셈이죠.

그렇다면 이것은 가능할까요?

```
/* void 형을 가리키는 포인터 */
#include <stdio.h>
int main() {
  void* a;

  return 0;
}
```

성공적으로 컴파일 해 본다면 아무런 오류가 뜨지 않는 다는 것을 알 수 있습니다. 왜 그럴까요? 일단, `void *a;` 의 경우 위에서 지적한 문제는 없다는 것을 알 수 있습니다. 왜냐하면 앞에서 `void a;` 의 경우 a 의 크기를 정할 수 없기 때문에 메모리 상에 a 를 위해 얼마나 많은 공간을 설정해 놓아야 하는지 모르지만, `void *a` 의 경우 '포인터' 이기 때문에 메모리 상에 8 바이트 만큼을 지정하게 됩니다. (앞에서 부터 강조해 왔던 이야기 이지만 64 비트 시스템에서 포인터의 크기는 8 바이트 입니다.) 즉, a 에는 어떠한 지점의 메모리의 주소 값이 들어가게 되는 것이지요.

그렇다면 `void* a` 포인터는 void 형의 변수의 메모리 주소를 가지게 될까요? 물론, 논리를 따지고 보면 맞지만 void 형 변수라는 것은 존재할 수 없기 때문에 void 형 포인터의 존재는 쓸모가 없어 보입니다. 하지만 사실 void 는 타입이 없기 때문에 거꾸로 생각해 보면어떠한 형태의 포인터의 값이라도 담을 수 있게 됩니다. 예를 들면

```
void *a;
double b = 123.3;

a = &b;
```

와 같이 말이죠. 다시 말해 a 는 순전히 오직 '주소값의 보관' 역할만 하게 되는 셈입니다.

```
/* b 의 값을 보려면 */
#include <stdio.h>
int main() {
  void *a;
  double b = 123.3;

  a = &b;

  printf("%lf", *a);
  return 0;
}
```

성공적으로 컴파일 하였다면

컴파일 오류

```
error C2100: 간접 참조가 잘못되었습니다.
```

와 같은 오류를 보게 됩니다. 이 오류가 발생하는 이유 역시 쉽게 알 수 있습니다. 왜냐하면 컴파일러는 *a 가 무엇을 말하는지 알 수 없거든요. 여태까지 *a 를 해석할 때 컴파일러는 a 가 가리키는 것의 타입을 보고 메모리 상에서 a 부터 얼마 만큼 읽어들어야 할 지 결정했는데 void a; 의 경우 메모리 상에서 얼마만큼 읽어들여야 할 지 모르기 때문입니다. 따라서 이는 다음과 같이 수정되어야 합니다.

```c
#include <stdio.h>
int main() {
  void *a;
  double b = 123.3;

  a = &b;

  printf("%lf", *(double *)a);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
123.300000
```

와 같이 잘 출력됨을 알 수 있습니다.

```c
printf("%lf", *(double *)a);
```

우리는 위 문장에서 형변환 을 이용하였습니다. 즉 단순히 주소값 만을 담고 있는 a 에게 (double *) 를 취함으로써, 컴파일러로 하여금 "이 포인터 a 가 담고 있는 주소값은 double 을 가리키는 주소값이라 생각해" 라고 말한 것이지요. 따라서 (double *)a 부분을 통해 컴파일러는 현재 a 가 가리키고 있는 곳의 주소값을 double 로 생각하게 되어 8 바이트를 읽어들이게 합니다.

void 형 포인터는 단순히 어떤 타입의 포인터의 주소 값도 편리하게 담을 수 있기 때문에 많은 부분에서 활용되고 있습니다. 예를 들어 다음과 같은 역할을 하는 함수를 생각해봅시다

어떠한 특정한 주소값으로 부터 1 바이트 씩 값을 읽어오는 함수

그렇다면 이 함수는에는 인자가 2 개 전달될 텐데, 일단 그 특정한 주소값을 가리키고 있는 포인터와, 얼마나 읽을지 int 형 변수 하나를 받아야 겠지요. 그런데, 인자로 전달될 '특정한 주소값을 가리키고 있는 포인터' 의 타입이 제각각 이라는 것이지요. 예를 들어서 int* 일 수 도 있고 double* 일 수 도 있지요.

따라서 우리는 순전히 주소값 만을 받기 위해서는 void 형 포인터를 사용하는 것이 바람직하다고 볼 수 있습니다. 물론 포인터 간의 형변환을 통해서 처리할 수 있지만 어떠한 형태의 포인터 주소값도 가능하다라는 의미를 살리기 위해서는 void 형 포인터를 이용하는 것이 바람직합니다.

```
/* 임의의 주소값 p 로 부터 byte 만큼 읽은 함수*/
#include <stdio.h>
int read_char(void *p, int byte);
int main() {
  int arr[1] = {0x12345678};

  printf("%x \n", arr[0]);
  read_char(arr, 4);
}
int read_char(void *p, int byte) {
  do {
    printf("%x \n", *(char *)p);
    byte--;

    p = (char *)p + 1;
  } while (p && byte);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
12345678
78
56
34
12
```

read_char 함수를 살펴봅시다. 무언가 여태까지 해온 것 보다 코딩 실력이 업그레이드 된 것 같은데 찬찬히 살펴 보면

```
do {
    printf("%x \n", *(char *)p);
    byte--;
```

```
    p = (char *)p + 1;
  } while (p && byte);
```

먼저 `p = (char *)p + 1;` 의 뜻 부터 생각해봅시다. `(char *)p` 는 '이 p 에 들어있는 값을 char 형 변수의 주소값이라 생각해!' 라는 의미 이지요. 그런데, 거기에 `+= 1` 을 했으므로 포인터의 덧셈이 행해지는데 컴퓨터는 p 를 'char 형 변수의 주소값' 이라 생각하고 있으므로 p 에 1 을 더하게 되면 주소값이 char 의 크기, 즉 1 만큼 늘어납니다.

아무튼 이와 같은 방법으로 p 의 주소값을 계속 1 씩 증가시키는데, 이 때 byte 의 값이 0 이되거나 `((char *)p)` 의 값이 0 (즉 NULL 일 때) while 문이 종료됩니다. 제가 `do while` 을 이용한 이유는 만일 동일한 조건문으로 while 문을 만들게 된다면 처음에 `((char *)p)++` 이 먼저 실행되기 때문에 p 부터 읽지 않고 `p + 1` 부터 읽게되는 불상사가 발생하기 때문에 이를 막기 위해 `do while` 문을 이용했습니다.

```
printf("%x \n", *(char *)p);
```

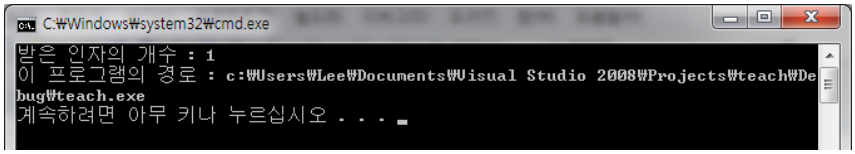
는 p 가 가리키는 주소값에 위치한 데이터 1 바이트 씩 16 진수로 출력하게 됩니다. 따라서 read_char 함수를 호출함을 통해 int 형 배열인 arr 의 원소를 1 바이트씩 읽게 되는 것이죠. 어떤 사람들은 그 결과가 12 34 56 78 순으로 출력해야 한다고 물을 수 있는데, 이는 '엔디안' 에 대한 개념이 없는 것이기 때문에 이 강좌를 잠시 보고 오시기 바랍니다.

간단히 말하자면 우리가 쓰는 대부분의 프로세서는 리틀 엔디안 방식으로 저장하기 때문에 낮은 자리수가 낮은 주소값을 가지게 됩니다. 즉, 낮은 자리수인 78 이 낮은 주소값인 앞쪽에 저장되게 되죠. 따라서 12 34 56 78 순이 아닌 78, 56, 34, 12 순으로 저장되는 것이 맞습니다. (그렇게 따지면 87, 65, 43, 21 순으로 나타나야 되지 않냐고 물을 수 있는데 저장의 단위가 바이트 이므로 한 바이트 내에서는 우리가 생각하는 순서대로 저장됩니다.)

메인 함수의 인자

```
/* main 함수의 인자라고?? */
#include <stdio.h>
int main(int argc, char **argv) {
  printf("받은 인자의 개수 : %d \n", argc);
  printf("이 프로그램의 경로 : %s \n", argv[0]);

  return 0;
}
```

성공적으로 컴파일 했다면



아마도 이 강좌를 보고 계신 여러분 중 일부는 위와 같은 메인 함수에 친숙하실 지도 모릅니다.

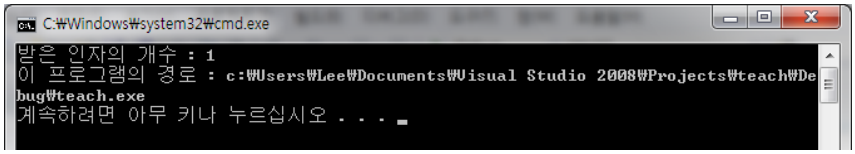
```
int main(int argc, char **argv)
```

보시다시피 main 함수가 인자를 받고 있습니다. 이게 도대체 무슨 일인가요. 다른 함수가 인자를 받는 것은 이해가 잘되는데 main 함수가 인자를 받다니. 도대체 누가 인자를 넣어 주고 있는 것일까요? 바로 운영체제에서 인자를 알아서 넣어주는 것입니다. 바로 위와 같이요.

일단 argc 는 main 함수가 받은 인자의 수 입니다. 그리고 argv 는 main 함수가 받은 각각의 인자들을 나타내죠. 프로그램을 실행하면 기본적으로 아무런 인자들을 넣지 않더라도 위와 같은 정보는 들어가게 됩니다. 즉, main 함수는 자신의 실행 경로를 인자로 받게 되죠. 그렇다면 다른 인자들도 넣을 수 있을까요? 한 번 해봅시다.

```
/* 인자를 가지는 메인 함수 */
#include <stdio.h>
int main(int argc, char **argv) {
  int i;
  printf("받은 인자의 개수 : %d \n", argc);

  for (i = 0; i < argc; i++) {
    printf("이 프로그램이 받은 인자 : %s \n", argv[i]);
  }

  return 0;
}
```

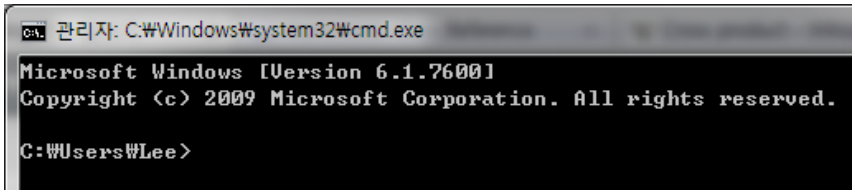
성공적으로 컴파일 했다면



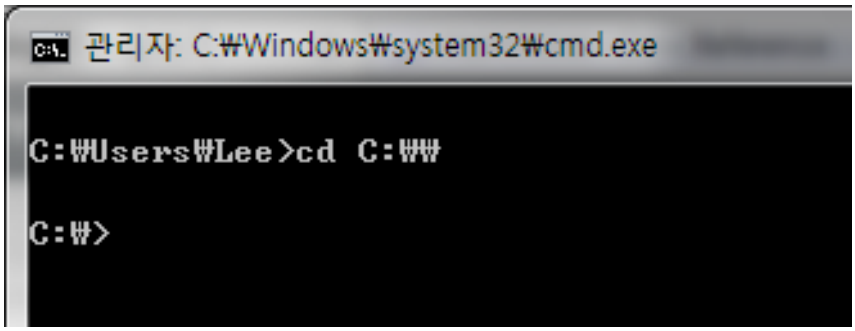
일단 우리는 프로그램을 임의의 개수의 인자를 받아 받은 인자들을 모두 출력하게 하였습니다. 그렇다면 프로그램에 직접 인자를 넣어 봅시다.

- 윈도우즈 XP 의 경우 : 시작 -> 실행 -> cmd
- 윈도우즈 Vista, 7 의 경우 : 시작 -> 하단에 '프로그램 밑 파일 검색' 에 cmd 라고 친다.

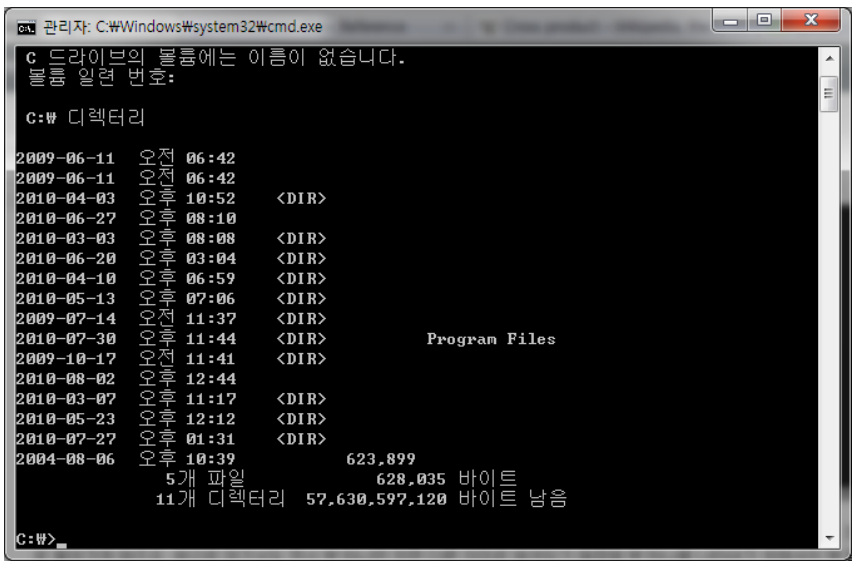
그렇다면 아래와 같은 모습을 보실 수 있습니다.



이것은 '명령 프롬포트' 라고 부르는 것인데 기존의 MS-DOS 와 유사합니다. (그러나 본질적으로 다릅니다) 우리는 여기서 윈도우즈 처럼 파일을 클릭하여 실행하는 것과는 달리 직접 명령어를 침으로써 파일을 실행시켜야 합니다. 그러기 위해선 우리가 원하는 파일이 어디있는지 알아야겠죠.



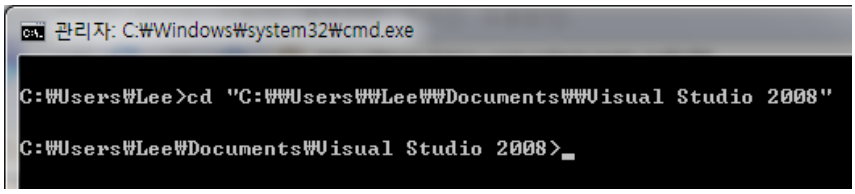
먼저 위와 같이 화면에 `cd C:\\` 을 씁니다. 이 명령어의 의미는 'C:\' 라는 경로로 들어가라 입니다. 즉 'cd' 의 의미는 지정하는 경로로 들어가게 해주죠. 참고로 여기서도 역시 \하나만 치면 다른 의미로 해석되기 때문에 \하나를 나타내기 위해서는 \를 두번 써야 합니다.



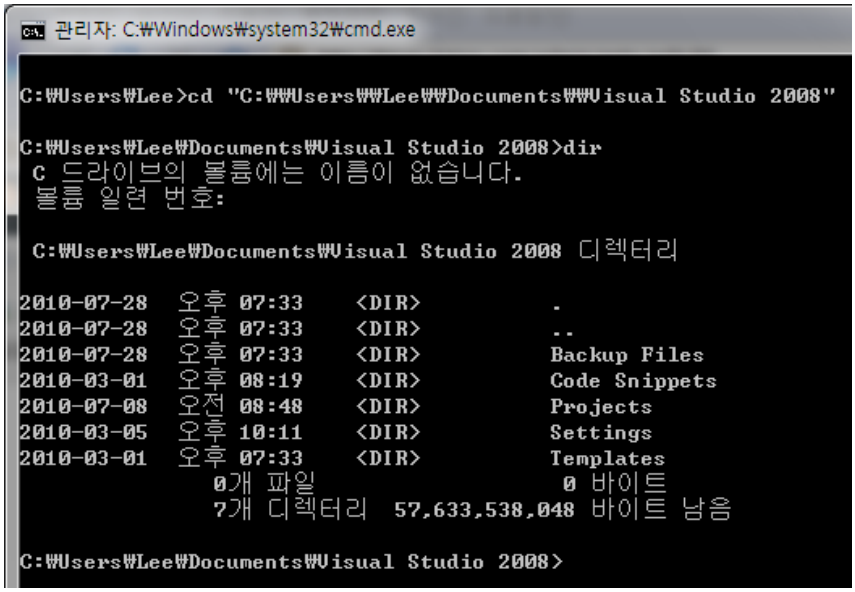
이제 화면에 `dir` 을 쳐봅시다. 'dir' 의 의미는 이 경로에 들어있는 폴더와 파일들을 보여주라는 의미입니다. 위 사진에서는 사생활보호 차원을 위해 `Program Files` 빼고 이름을 모두 가렸습니다. 이 때, 어떤 것은 왼쪽에 `<DIR>` 이라고 나오고 어떤 것은 없는 것을 볼 수 있는데, `<DIR>` 이란 것은

'폴더' 와 같은 뜻으로 파일이 아니라는 것입니다. 반면에 <DIR> 이 없는 것은 파일이 되겠지요. 아마 프로그램을 C 드라이브에 깔았기 때문에 아마 모든 파일은 동일한 경로에 있을 것입니다.

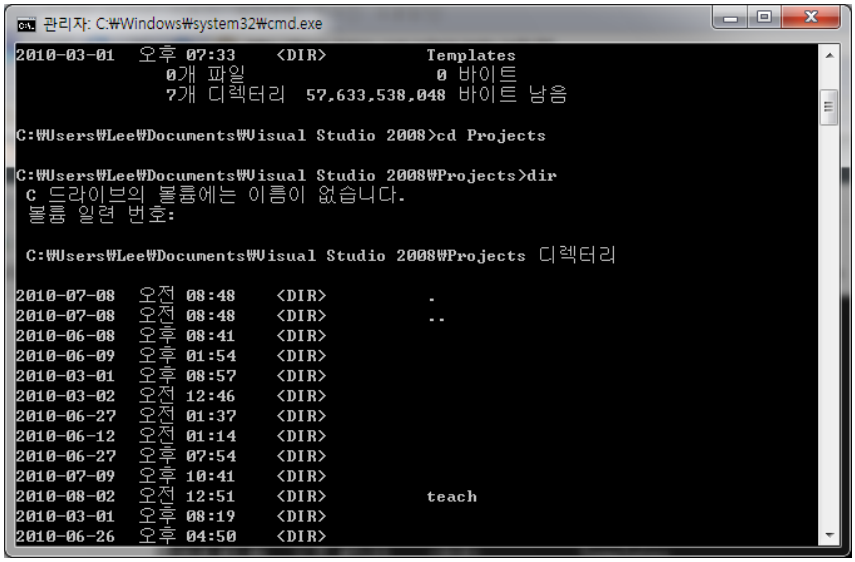
위와 같은 방식으로 우리의 파일을 찾는 일만 남았습니다.



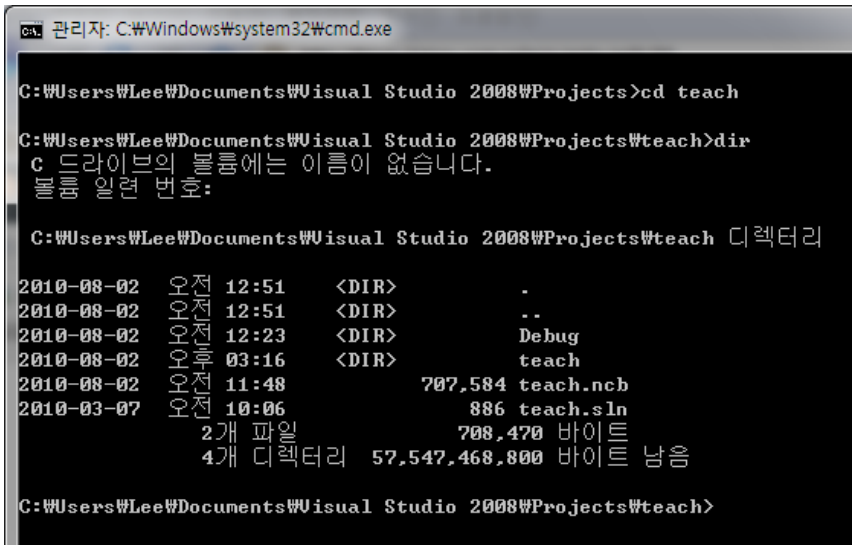
우리가 원하는 파일은 아마 C:\Users\Lee\Documents\Visual Studio 2008 에 있습니다. 이를 cd 명령어로 다 치면 됩니다. 이 때, 중간에 띄어쓰기가 있으므로 큰 따옴표로 묶어주어야 합니다. 즉 cd "C:\\Users\\Lee\Documents\\Visual Studio 2008" 처럼 말이죠. 그렇지 않고 cd C:\\Users\\Lee\Documents\\Visual Studio 2008 로 쓴다면 컴퓨터는 cd C:\\Users\\Lee\Documents\\Visual 로 인식합니다.



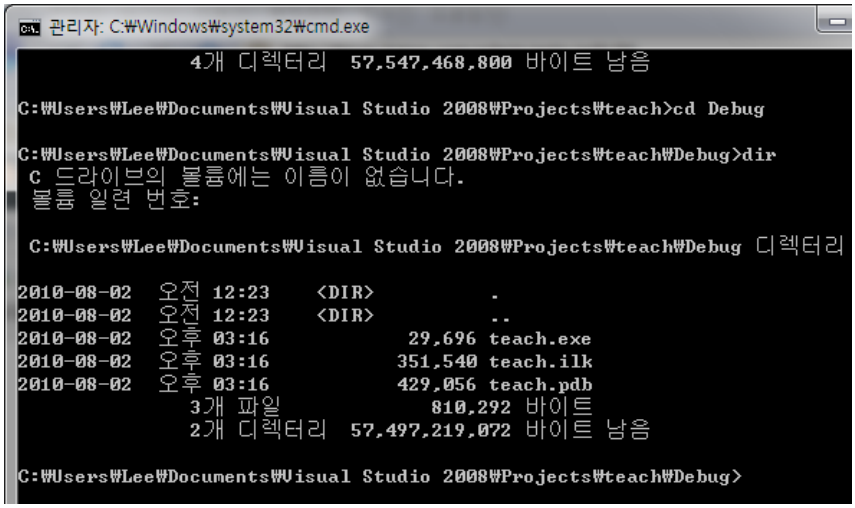
이제 Project 폴더로 들어가보겠습니다. 단순히 cd Projects 라고 치면 됩니다.



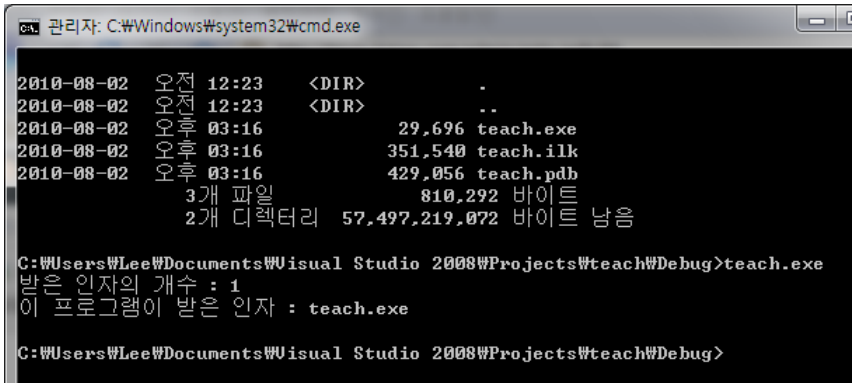
오. 저의 프로젝트인 'teach' 가 보이네요. 여러분과 저와의 프로젝트 이름이 다를 수 있으니 각기 맞는 프로젝트로 들어가시면 됩니다. 여기서도 물론 폴더 이름에 띄어쓰기가 있다면 큰따옴표로 묶어주는 것을 잊지 마세요.



이제, 파일들을 쭉 보면 Debug 라는 폴더와 teach 라는 폴더가 있는데, teach 폴더에는 우리의 소스 코드가, Debug 폴더에는 만들어진 실행 파일이 있습니다. 그렇다면 우리는 어디로 가야 할까요? 네, Debug 로 갑시다.



우와 그렇다면 위와 같이 teach.exe 를 보실 수 있습니다. 이제 teach.exe 를 침으로써 위 프로그램을 실행할 수 있습니다.

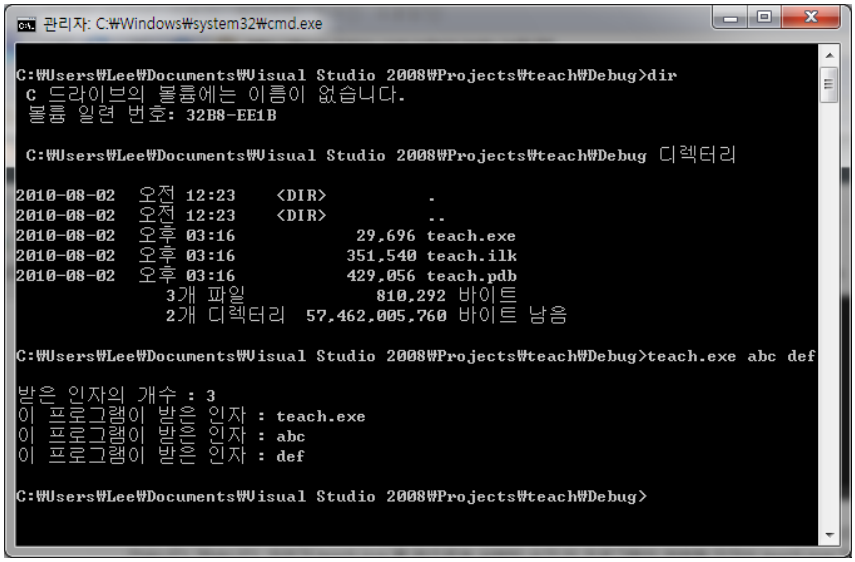


우왕. 잘 실행되는 군요. 일단 신기한 점은 이 프로그램이 받은 인자가 더이상 그 프로그램의 경로가 들어가지 않고 teach.exe 가 들어갔습니다. 맞습니다. 우리가 teach.exe 를 침으로써 실행한 순간 이 프로그램의 첫번째 인자는 teach.exe 가 됩니다.

만일 우리가 이 프로그램을 "C:\\Users\\Lee\\Documents\\Visual Studio 2008\\Projects\\teac
라고 쳐서 실행하였다면 인자가 C:\\Users\\Lee\\Documents\\Visual Studio 2008\\Projects\\te
가 되겠지요. [1)]

그렇다면 다른 인자들을 넣어봅시다. 이는 간단합니다. 프로그램 이름 뒤에 다른 것들을 써주면 되죠. 예를 들어

1) 사실 이 부분에 대해서는 딱히 정해진 것은 없습니다. 따라서 argv 의 첫 번째 인자는 보통 무시하게 됩니다.



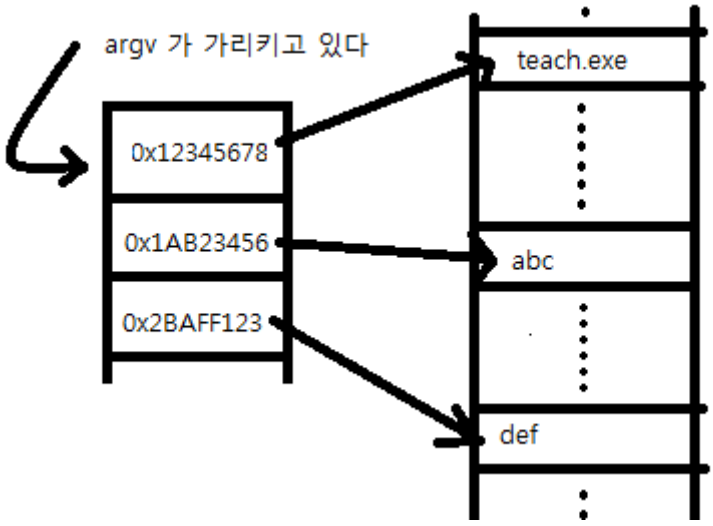
`teach.exe abc def` 라고 쓰게 된다면 `teach.exe` 가 첫번째 인자, `abc` 가 두번째 인자, `def` 가 세번재 인자가 되고 받은 인자의 수는 3 이 되지요. 어때요. 쉽지요?

그런데 여태까지 배운 내용을 잘 숙지하신 분이라면 다음과 같은 질문을 하실 수 있습니다.

`main` 함수의 두번째 인자 말이에요, `char **` 인데 제 기억에 이차원 배열을 전달하기 위해서는 `char (*argv)[5]` 와 같이 반드시 크기를 명시해 주어야 하는데 여기서는 단순히 `char**` 로 해놓고 어떻게 그리 잘 작동하는지요?

그 이유는 간단합니다. `char**` 은 `(char *)` 형 배열을 가리키는 포인터 이지요. 즉, 포인터의 배열 입니다. (배열 포인터가 절대로 아닙니다) `int arr[4]` 라는 배열을 가리키는 포인터가 `int *` 형인 것 (여기서는 `arr` 이겠네요) 처럼 `char *arr[5];` 를 가리키는 포인터의 형은 `char**` 이 되겠지요.

즉, 다음과 같은 꼴이 되겠지요.



즉 `argv` 는 포인터들의 배열을 가리키고 있고, 그 포인터 배열에서의 각각의 원소, 즉 포인터들은

인자로 전달된 문자열들을 가리키고 있습니다. 이 때, 이 문자열들은 메모리의 다른 공간에 보관되어 있겠죠.

따라서 우리는 `argv[i]` 를 통해 특정한 인자의 문자열에 저장된 주소값을 나타낼 수 있게 됩니다.

그럼 이상으로 이번 강좌를 마치도록 하겠습니다. 이번 강좌는 다음에 배울 동적 메모리 할당에 밑바탕이 되는 정보 이니 절대로 잊지 마시기 바랍니다.

생각해 보기

문제 1

메인함수의 인자를 활용한 계산기를 만들어보세요. 예를 들어서

```
calc.exe 5 + 10
```

을 치면 15 가 나오게 하면 되지요.

이 때, `5, +, 10` 은 모두 다른 인자로 봐야하겠죠. 기초적인 단계 이므로 연산자는 하나만 써도 된다고 합시다.참고로 인자는 모두 문자열 형태로 오기 때문에 문자열로 된 수를 `int` 형으로 바꾸는 작업이 필요할 것입니다.

18

동적 메모리 할당 (dynamic memory allocation)

안녕하세요. 여러분. 정말 멀리 달려 온 것 같네요. 벌써 제 20 장을 지나가고 있습니다. 물론 강의 수로 따지면 더 많죠.

아마도 여러분은 프로그램을 만들면서 다음과 같은 문제에 봉착했던 적들이 많았었을 것입니다.

배열의 크기를 자유 자재로 다룰 수 있으면 얼마나 좋을까?

맞습니다. 우리가 배열을 정할 때 그 크기는 언제나 **컴파일 시간에 확정** 되어 있어야 합니다. 즉 컴파일러가 배열의 크기를 추측할 필요 없이 명확하게 나타나 있어야 된다는 것이지요. 하지만 이는 정말 고역스러운 일이 아닙니다. 예를 들어 우리가 컴퓨터로 부터 학생들의 수학 점수를 입력 받아 평균을 내는 프로그램을 만든다고 해봅시다. 각 학급 마다 학생들의 수가 모두 다르기 때문에 배열의 크기를 명확하게 정할 수 없게 됩니다. 따라서 보통 이 경우 배열을 '충분히 크게' 잡게 되는데 이렇게 된다면 메모리가 낭비되는 경우가 허다하게 발생합니다.

컴퓨터에서 낭비란 곧 비효율적인 프로그램을 의미하는 것이지요. 이렇게 쓸데 없이 낭비되는 자원을 막기 위해 '학생 수' 를 입력 받고 그 학생 수 만큼 배열의 크기를 지정하면 얼마나 좋을까요. 하지만 놀랍게도 이렇게 할 수 있는 방법이 있습니다.

바로 **동적 메모리 할당** 이라는 방법 입니다. 이 것은 말그대로 동적으로 메모리를 할당 합니다. 여기서 '동적' 이란 말은 딱 정해진 것이 아니라 가변적으로 변할 수 있다는 말이지요. 또한 메모리를 '할당' 한다는 이야기는 역시 우리가 배열을 정의하면 배열에 맞는 메모리의 특정한 공간이 배열을 나타내는 것 처럼 메모리의 특정한 부분을 사용할 수 있게 됩니다.참고적으로 아마 다 아시겠지만 할당되지 않는 메모리는 절대로 사용할 수 없습니다.

도대체 어떻게 그런 일이 가능할까요.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
```

```
  int SizeOfArray;
  int *arr;

  printf("만들고 싶은 배열의 원소의 수 : ");
  scanf("%d", &SizeOfArray);

  arr = (int *)malloc(sizeof(int) * SizeOfArray);
  // int arr[SizeOfArray] 와 동일한 작업을 한 크기의 배열 생성

  free(arr);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
만들고 싶은 배열의 원소의 수 : 5
```

일단 위 예제를 통해서는 정말 우리가 원하는 크기의 배열이 생겼는지는 모르겠지만 일단 위 소스코드 부터 파헤쳐봅시다.

```
printf("만들고 싶은 배열의 원소의 수 : ");
scanf("%d", &SizeOfArray);
```

먼저 우리는 위 과정을 통해서 우리가 원하고자 하는 `int` 배열의 원소의 개수를 입력 받았습니다. 그리고,

```
arr = (int *)malloc(sizeof(int) * SizeOfArray);
```

두둥. 바로 이 녀석이 우리가 원하는 작업을 해주는 역할을 합니다. 이 함수의 이름은`malloc` 이며 **memory allocation** 의 약자 입니다.

이 함수는 `<stdlib.h>` 에 정의되어 있기 때문에 `#include <stdlib.h>` 를 추가해주어야 합니다.

이 함수는 인자로 전달된 크기의 바이트 수 만큼 메모리 공간을 만듭니다. 즉 메모리 공간을 할당하게 되는 것이지요. 우리가 원소의 개수가 `SizeOfArray` 인 `int` 형 배열을 만들기 위해서는 당연히 `(int 의 크기) * (SizeOfArray)` 가 되겠지요. 이 때, `int` 타입의 크기를 정확하게 알기 위해서 `sizeof` 키워드를 사용하게 됩니다. `sizeof` 는 이 타입의 크기를 알려줍니다. 따라서 `sizeof(int) * SizeOfArray` 를 인자로 전달해 주면 됩니다.

이 함수가 리턴하는 것은 자신이 할당한 메모리의 시작 주소를 리턴하게 됩니다. 이 때, 리턴형이 `(void *)` 형이므로 우리는 이를 `(int *)` 형으로 형변환 하여 `arr` 에 넣어주기만 하면 됩니다. 이렇게 보니 마치 `malloc` 함수가 공원에서 돗자리를 까는 역할을 하는 것과 같네요. 사람이 바글바글한 공원에서 `malloc` 함수는 '원하는 크기의 돗자리' 를 깔아주고 이 돗자리로 사람들이 올 수 있도록 손을 흔들어주는 역할을 하는 것과 같습니다.

따라서 `arr` 에는 `malloc` 이 할당해준 메모리를 이용할 수 있게 됩니다. 즉, `arr[SizeOfArray]` 만큼을 사용할 수 있게 되죠.

```
free(arr);
```

그리고 마지막에`free` 는 우리가 할당받은 다 쓰고 난 후에 메모리 영역을 다시 컴퓨터에게 돌려주는 역할을 합니다. 이를 **해제(free)** 한다 그러는데 이 `free` 를 제대로 하지 않게 된다면 딱히 사용하지도 않는 메모리를 쓸데없이 자리만 차지하게 되겠지요.

이렇게 `free` 를 제대로 하지 않아 발생되는 문제를 **메모리 누수(memory leak)** 이라고 합니다. 이는 마치 공원에 돗자리를 깔아놓고 그대로 놓고 집에 가는 것과 똑같은 일입니다. (이런 일이 반복된다면 나중에 다시 왔을 때 공원에는 돗자리를 놓을 수 있는 공간이 하나도 없겠죠?)

malloc 은 어디에 할당할까?

우리가 이전에 17 강에서 메모리 구조에 대해 배울 때 메모리에는 다음과 같은 구조들이 있다는 것을 배웠습니다.

이 때 다른 부분은 모두 설명하였는데 오직 **힙(Heap)** 에 대해서만 언급을 안했다는 것을 기억 하실 것입니다. 자, 이제 드디어 힙의 정체를 알 수 있는 시간입니다. 스택 이나, 데이타 영역, Read Only Data 부분은 당연하게도 `malloc` 함수가 결코 건드릴 수 없는 부분 입니다. 이 부분의 크기는 반드시 컴파일 때에 100% 추호의 의심의 여지도 없이 정해져야 합니다.

하지만 힙의 경우 다릅니다. 메모리의 힙 부분은 사용자가 자유롭게 할당하거나 해제할 수 있습니다. 따라서 우리의 `malloc` 함수도 이 힙을 이용하는 것입니다. 우리가 만들어낸 `arr` 은 힙에 위치하고 있습니다.

힙은 할당과 해제가 자유로운 만큼 제대로 사용해야 합니다. 만일 힙에 할당은 하였는데 해제를 하지 않았다면 공간이 낭비되겠지요. 다른 메모리 부분의 경우 컴퓨터가 알아서 처리하기 때문에 문제가 발생할 여지가 적지만 힙은 인간이 다루는 만큼 철저히 해야 합니다.

```c
/* 동적 할당의 활용 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
```

```
  int student;  // 입력 받고자 하는 학생 수
  int i, input;
  int *score;   // 학생 들의 수학점수 변수
  int sum = 0;  // 총점

  printf("학생의 수는? : ");
  scanf("%d", &student);

  score = (int *)malloc(student * sizeof(int));

  for (i = 0; i < student; i++) {
    printf("학생 %d 의 점수 : ", i);
    scanf("%d", &input);

    score[i] = input;
  }

  for (i = 0; i < student; i++) {
    sum += score[i];
  }

  printf("전체 학생 평균 점수 : %d \n", sum / student);
  free(score);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
학생의 수는? : 3
학생 0 의 점수 : 100
학생 1 의 점수 : 90
학생 2 의 점수 : 95
전체 학생 평균 점수 : 95
```

와 같이 나옵니다.

```
score = (int *)malloc(student * sizeof(int));
```

먼저 위 부분을 통해 원소의 개수가 `student` 인 `int` 형 배열을 생성하였죠. 따라서 우리는 `score` 을 `int score[student]` 로 한 것 마냥 사용할 수 있게 됩니다.

```
for (i = 0; i < student; i++) {
  printf("학생 %d 의 점수 : ", i);
```

```
  scanf("%d", &input);

  score[i] = input;
}

for (i = 0; i < student; i++) {
  sum += score[i];
}
```

따라서 위와 같이 score 에 원소를 입력 받고 그 원소들을 모두 더해 평균을 구하게 됩니다. 어때요. 간단하지요.

2 차원 배열의 동적 할당

그렇다면 좀더 높은 난이도의 문제에 도전해봅시다. 2 차원 배열을 동적으로 할당할 수 있을까요? 물론 가능합니다. 여러분은 지금 머리속으로 마구 어떻게 할지 생각하고 있으실 텐데 의외로 간단합니다. 바로 포인터 배열을 이용하면 됩니다.

포인터 배열이라 함은 이전에도 이야기 했었지만 배열의 각 원소들이 모두 포인터 인 것이지요. 따라서, 각 원소들이 다른 일차원 배열들을 가리킬 수 있습니다. 따라서, 이 배열은 2 차원 배열이 되겠지요. 따라서 우리가 해야할 일은 먼저 포인터 배열을 동적으로 할당한 뒤에 다시 포인터 배열의 각각의 원소들이 가리키는 일차원 배열을 다시 동적으로 할당해 주면 됩니다.

그럼 이를 실행에 옮기도록 하겠습니다.

```
/* 2 차원 배열의 동적 할당 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  int i;
  int x, y;
  int **arr;  // 우리는 arr[x][y] 를 만들 것이다.

  printf("arr[x][y] 를 만들 것입니다.\n");
  scanf("%d %d", &x, &y);

  arr = (int **)malloc(sizeof(int *) * x);
  // int* 형의 원소를 x 개 가지는 1 차원 배열 생성

  for (i = 0; i < x; i++) {
    arr[i] = (int *)malloc(sizeof(int) * y);
  }

  printf("생성 완료! \n");
```

```
  for (i = 0; i < x; i++) {
    free(arr[i]);
  }
  free(arr);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
arr[x][y] 를 만들 것입니다.
3
5
생성 완료!
```

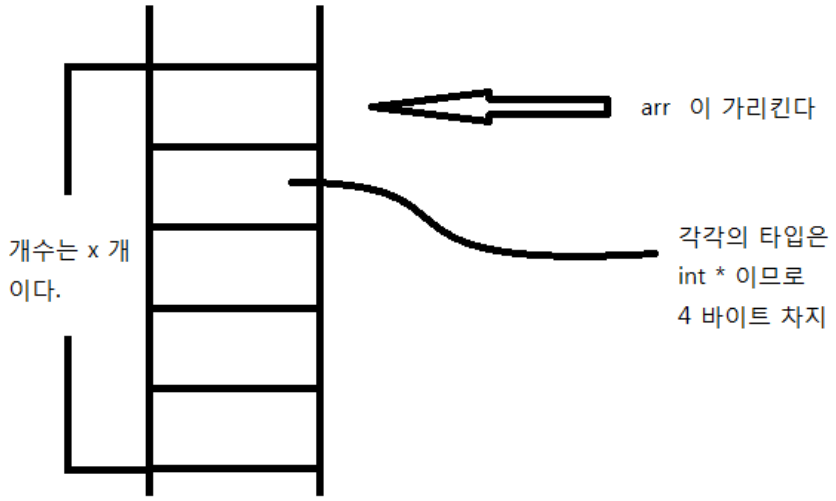
흠. 일단 잘 생성은 된 것 같기는 한데, 한 번 차근 차근 소스를 따라가 봅시다.

```
int **arr;  // 우리는 arr[x][y] 를 만들 것이다.
```

일단 `int **arr` 부터 봅시다. 만일 `int array[3];` 이란 배열을 만들었다면 `array` 의 형은 무엇일까요. 네 맞습니다. `int *` 입니다. 그렇다면 `int * arr[10];` 이란 배열을 만들었다면 `arr` 의 형은? 네. `int **arr` 이죠. 따라서 우리는 `int **arr;` 과 같이 선언하였습니다.

```
arr = (int **)malloc(sizeof(int *) * x);
```

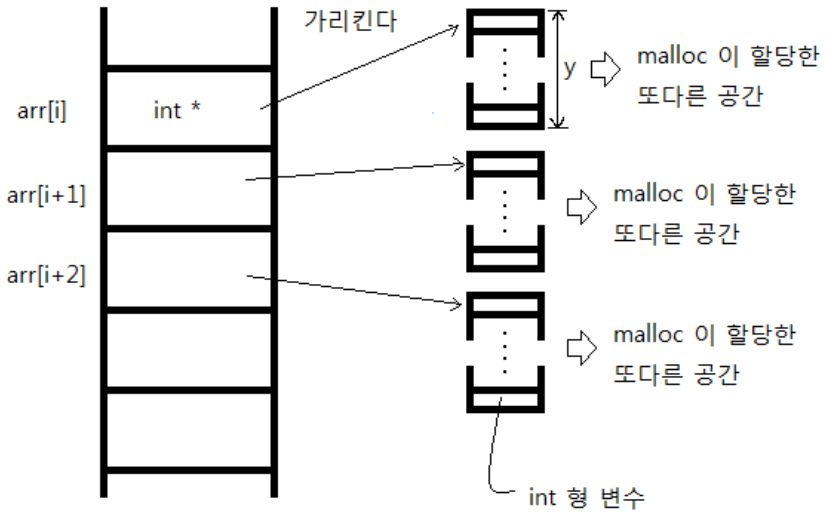
따라서 위와 같이 `int *` 형 배열을 동적 할당 할 수 있었습니다. 위 과정을 거치게 되면 `arr` 은 다음과 같은 모습일 것입니다.



자. 그럼 arr 배열의 각각의 원소들은 int * 형 이므로 다른 int 배열을 가리키기를 갈망하고 있을 것입니다. 우리는 그 욕구를 해소 시켜 주어야 겠죠. 따라서 각각의 원소들에 대해 원하는 메모리 공간을 짝지어 줍시다.

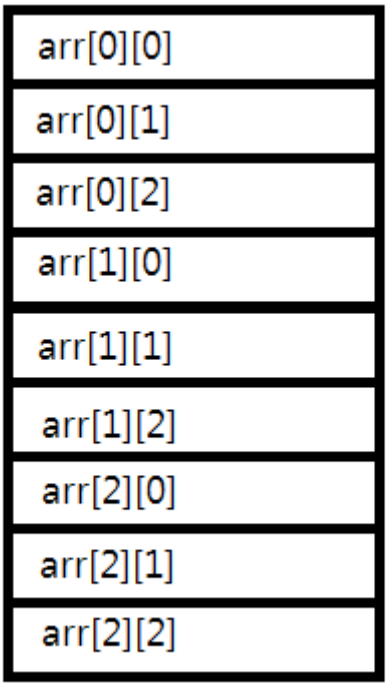
```
for (i = 0; i < x; i++) {
  arr[i] = (int *)malloc(sizeof(int) * y);
}
```

각각의 원소들에 대해 메모리 공간을 할당하고 있습니다. arr[i] 는 malloc 이 정의한 또다른 공간을 가리키겠네요.



따라서 arr 의 하나의 원소가 크기가 y 인 배열을 가리키고 있는데 arr 의 원소가 x 개 이므로

전체적으로 보았을 때 총 `x * y` 배열을 가지는 셈입니다. 하지만 이렇게 만들어진 배열은 정확히 말해 2 차원 배열이라 말하기는 힘듧니다. 왜냐하면 배열은 모름지기 메모리에 연속적으로 있어야 하기 때문이죠. 예를 들어 이전 강의의 사진을 잠깐 가져오면



와 같이 말이지요. 하지만 우리가 만든 배열은 `arr` 의 원소들이 가리키는 메모리 공간이 연달아 존재한다고 보장할 수 없습니다. 또한 한 가지 재미있는점은 우리가 만든 '2 차원 배열 처럼 생긴' 포인터 배열은 2 차원 배열과는 달리 함수의 인자로 손쉽게 넘길 수 있습니다. 예를 들면

```
int array(int **array);
```

처럼 말이지요. `array(arr);` 을 하게 되면 우리가 만든 배열을 함수에 넘길 수 있게 됩니다. 이게 가능한 이유는 사실 우리가 만든 배열은 1 차원 배열들이지 2 차원 배열이 아니기 때문입니다. `arr` 은 단순히 `int *` 형 원소들을 가지는 1 차원 배열 이지요. 1 차원 배열을 함수의 인자로 넘겨줄 때에는 크기를 써 주지 않아도 되지 않았습니까. 사실 `main` 함수의 인자로 전달되는 argv 역시 이와 같은 성격을 띕니다.

그렇다고 해서 2 차원 배열의 성질을 잃어버리는 것은 아닙니다. 이 배열도 2 차원 배열 처럼 `arr[3][4]` 과 같이 원소에 접근할 수 있습니다 (그렇기 때문에 우리가 만든 이 배열을 2 차원 배열이라 부르는 것입니다). 왜냐하면 `arr[3][4]` 는 `*(*(arr + 3)+4)` 로 해석되는데, `*(arr + 3)` 을 통해 `arr` 의 네번째 원소에 접근하게 되고 `*(arr + 3)` 은 자신이 가리키는 `int` 형 배열의 주소값을 의미하므로 `+ 4` 를 하면 `int` 형 배열의 5 번째 원소에 접근하는 것과 같습니다.

아무튼 이와 같은 방법으로 2 차원 배열 (사실은 다르지만 이렇게 부르겠습니다) 를 생성하였습니다. 우리가 이 배열을 힙에 할당하였으면 사용이 끝났으면 역시 되돌려 주어야 하겠죠. 해제하는 순서는

할당하는 순서와 정 반대로 하면 됩니다. 즉, `arr[i]` 들이 가리키고 있던 `int` 배열들을 해제한 후, `arr` 을 해제하면 되겠지요. 만일 `arr` 을 먼저 해제하면 `arr[i]` 들이 메모리 상에서 사라지게 되므로 `arr[i]` 들이 가리키고 있던 `int` 배열들을 해제할 수 없게 되므로 오류가 나게 됩니다.

```c
/* 2 차원 배열 동적 할당의 활용 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  int i, j, input, sum = 0;
  int subject, students;
  int **arr;
  // 우리는 arr[subject][students] 배열을 만들 것이다.

  printf("과목 수 : ");
  scanf("%d", &subject);

  printf("학생의 수 : ");
  scanf("%d", &students);

  arr = (int **)malloc(sizeof(int *) * subject);

  for (i = 0; i < subject; i++) {
    arr[i] = (int *)malloc(sizeof(int) * students);
  }

  for (i = 0; i < subject; i++) {
    printf("과목 %d 점수 --------- \n", i);

    for (j = 0; j < students; j++) {
      printf("학생 %d 점수 입력 : ", j);
      scanf("%d", &input);

      arr[i][j] = input;
    }
  }

  for (i = 0; i < subject; i++) {
    sum = 0;
    for (j = 0; j < students; j++) {
      sum += arr[i][j];
    }
    printf("과목 %d 평균 점수 : %d \n", i, sum / students);
  }

  for (i = 0; i < subject; i++) {
    free(arr[i]);
  }

  free(arr);
```

```
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
과목 수 : 3
학생의 수 : 2
과목 0 점수 ---------
학생 0 점수 입력 : 90
학생 1 점수 입력 : 100
과목 1 점수 ---------
학생 0 점수 입력 : 80
학생 1 점수 입력 : 70
과목 2 점수 ---------
학생 0 점수 입력 : 60
학생 1 점수 입력 : 100
과목 0 평균 점수 : 95
과목 1 평균 점수 : 75
과목 2 평균 점수 : 80
```

와 같이 나옵니다. 대성공이군요!

```
int **arr;
```

위 예제에서 우리는 과목별 학생의 점수를 보관하기 위해 이차원 배열을 사용하였습니다. 즉 `arr[subject][students]` 를 만든 것이지요. 이를 위해

```
arr = (int **)malloc(sizeof(int *) * subject);

for (i = 0; i < subject; i++) {
  arr[i] = (int *)malloc(sizeof(int) * students);
}
```

를 통해 `arr[subject][students]` 를 만들 수 있었습니다. 따라서 이제 과목별 학생의 점수를

```
for (i = 0; i < subject; i++) {
  printf("과목 %d 점수 --------- \n", i);
```

```
  for (j = 0; j < students; j++) {
    printf("학생 %d 점수 입력 : ", j);
    scanf("%d", &input);

    arr[i][j] = input;
  }
}
```

로 얻었습니다. arr 은 사실 2 차원 배열은 아니지만 2 차원 배열과 똑같이 행동하므로 arr[i][j] 와 같은 문장도 맞게 되지요. arr[i][j] 를 i 행 j 열에 위치한 값이라 생각해도 무방합니다.

```
for (i = 0; i < subject; i++) {
  sum = 0;
  for (j = 0; j < students; j++) {
    sum += arr[i][j];
  }
  printf("과목 %d 평균 점수 : %d \n", i, sum / students);
}
```

이제 값을 모두 입력받았다면 각 과목별 평균을 내면 되는데 이는 간단히 위와 같은 for 문으로 해결할 수 있었습니다.

```
for (i = 0; i < subject; i++) {
  free(arr[i]);
}

free(arr);
```

마지막으로 할당 받은 메모리의 사용이 끝났기 때문에 해제해야 하는데 이는 이전에 설명했던 예제와 동일하게 하면 됩니다.

```
/* 할당한 (2 차원 배열 처럼 생긴) 배열 전달하기 */
#include <stdio.h>
#include <stdlib.h>

void get_average(int **arr, int numStudent, int numSubject);

int main(int argc, char **argv) {
  int i, j, input, sum = 0;
  int subject, students;
  int **arr;
  // 우리는 arr[subject][students] 배열을 만들 것이다.

  printf("과목 수 : ");
```

```
  scanf("%d", &subject);

  printf("학생의 수 : ");
  scanf("%d", &students);

  arr = (int **)malloc(sizeof(int *) * subject);

  for (i = 0; i < subject; i++) {
    arr[i] = (int *)malloc(sizeof(int) * students);
  }

  for (i = 0; i < subject; i++) {
    printf("과목 %d 점수 --------- \n", i);

    for (j = 0; j < students; j++) {
      printf("학생 %d 점수 입력 : ", j);
      scanf("%d", &input);

      arr[i][j] = input;
    }
  }

  get_average(arr, students, subject);

  for (i = 0; i < subject; i++) {
    free(arr[i]);
  }
  free(arr);

  return 0;
}
void get_average(int **arr, int numStudent, int numSubject) {
  int i, j, sum;

  for (i = 0; i < numSubject; i++) {
    sum = 0;
    for (j = 0; j < numStudent; j++) {
      sum += arr[i][j];
    }
    printf("과목 %d 평균 점수 : %d \n", i, sum / numStudent);
  }
}
```

성공적으로 컴파일 했다면

실행 결과

```
과목 수 : 2
```

```
학생의 수 : 3
과목 0 점수 ---------
학생 0 점수 입력 : 100
학생 1 점수 입력 : 90
학생 2 점수 입력 : 80
과목 1 점수 ---------
학생 0 점수 입력 : 70
학생 1 점수 입력 : 80
학생 2 점수 입력 : 90
과목 0 평균 점수 : 90
과목 1 평균 점수 : 80
```

와 같이 나옵니다. 다른 부분은 모두 똑같으므로 함수만 살펴봅시다.

```
void get_average(int **arr, int numStudent, int numSubject)
```

일단 `void` 형이고 `int **arr` 와 `numStudent, numSubject` 를 인자로 받고 있습니다. 앞에서 설명 했지만 `arr` 은 2 차원 배열 처럼 행동함에도 불구하고 사실은 단순히 원소가 `int *` 형인 배열이기 때문에 (1 차원 배열의 경우 단순히 배열의 타입에 `*` 만 붙이면 된다는 사실은 다 알고계시죠?) 위와 같이 `int **arr` 로 기존의 2 차원 배열 처럼 열의 개수에 대한 정보가 없어도 됩니다. (2 차원 배열의 경우 `int (*arr)[3]`과 같이 열에 관한 정보가 있어야 함)

물론 함수 내부에서 총 학생의 명수와 총 과목의 개수를 알아야 하므로 위와 같이 `numStudent` 와 `numSubject` 를 넣어주었지만요. 이제 여러분은 C 언어의 대부분을 배웠다고 해도 무방합니다만, 아직 몇 가지 재미있는 것들이 남아있으니 다음 강좌가 나올 때 까지 생각해 볼 문제나 풀어보세요 :)

생각해보기

문제 1

위 성적 프로그램을 개량하여 학생별 평균을 내어 학생의 등수를 출력하는 프로그램을 만들어보세요 (난이도 : 下)

문제 2

동적으로 할당된 배열의 크기를 다시 바꾸는 프로그램을 만들어보세요.즉 p 가 이미 원소가 10 인 동적으로 할당된 배열을 가리키고 있었는데 예상치 못하게 원소 5 개를 더 추가하려면 어떻게 해야 할까요. (난이도 : 中)

구조체의 동적 할당과 메모리 관리 함수

안녕하세요 여러분. 메모리에 관해서 두 번째 이야기를 풀어 나가려고 합니다. 원래 메모리 동적 할당은 강의 한 개로 끝내려고 했는데 코이치 님이 무언가 조금 모자라다는 듯한 느낌이 든다고 하셔서 두 개의 강의로 이어 나가려고 합니다. 물론 동적 할당에 관한 기본 개념은 지난 강좌에서 모두 다루었지만 조금 보충 설명과 함께 새로운 것들을 이야기 하고자 합니다.

구조체 동적 할당

```c
#include <stdio.h>
#include <stdlib.h>
struct Something {
  int a, b;
};
int main() {
  struct Something *arr;
  int size, i;

  printf("원하시는 구조체 배열의 크기 : ");
  scanf("%d", &size);

  arr = (struct Something *)malloc(sizeof(struct Something) * size);

  for (i = 0; i < size; i++) {
    printf("arr[%d].a : ", i);
    scanf("%d", &arr[i].a);
    printf("arr[%d].b : ", i);
    scanf("%d", &arr[i].b);
  }

  for (i = 0; i < size; i++) {
    printf("arr[%d].a : %d , arr[%d].b : %d \n", i, arr[i].a, i, arr[i].b);
  }

  free(arr);

  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
원하시는 구조체 배열의 크기 : 2
arr[0].a : 1
arr[0].b : 2
arr[1].a : 3
arr[1].b : 4
arr[0].a : 1 , arr[0].b : 2
arr[1].a : 3 , arr[1].b : 4
```

와 같이 나옵니다.

저의 구조체 강좌를 여태까지 잘 보신 분들은 잘 아시겠지만 구조체 역시 특별하게 생각해야 될 것이 아니라 '사용자가 만든 하나의 데이터 타입' 이라고 보시면 된다고 했습니다. 다시 말해 구조체도 `int` 처럼 사용할 수 있다는 것이지요. 따라서 구조체 배열을 `malloc` 을 이용하여 지지고 볶는 일은 전혀 이상할 것이 없는 행동 입니다.

```
struct Something *arr;
```

일단 1 차원 구조체 배열을 가리키기 위한 `arr` 을 선언하였습니다. `int` 형 배열을 만들기 위해 `int *arr;` 이라 했던 것과 정확히 일치 합니다.

```
arr = (struct Something *)malloc(sizeof(struct Something) * size);
```

이제 `malloc` 함수를 이용하여 `arr` 을 위한 공간을 할당해줍니다. 이에 필요한 크기는 당연히도 `sizeof(struct Something) * size` 입니다. 만일 `sizeof` 대신에 구조체의 실제 크기를 계산해서 더하시는 분이 있는데 이는 오류를 발생 시킬 수 있습니다. 예를 들어 위 `Somehting` 구조체의 경우 1 개당 8 바이트를 차지한다고 볼 수 있는데 사실 그렇지 않을 수 도 있습니다.

물론 위 경우는 조금 특별하지만 예를 들어 구조체의 크기가 10 바이트일 경우 컴퓨터가 **더블워드 경계(double word boundary)** 에 놓음으로 속도를 향상시키는 경우가 있는데 이 경우 구조체의 크기는 12 바이트로 간주될 수 있습니다. 사실 자세한 내용은 여기서 생략하기로 하고 아무튼 기억해야 할 점은 언제나 `sizeof` 를 사용해야 한다는 점입니다. 무턱대고 크기를 추정하지 맙시다!

```
for (i = 0; i < size; i++) {
  printf("arr[%d].a : ", i);
  scanf("%d", &arr[i].a);
  printf("arr[%d].b : ", i);
  scanf("%d", &arr[i].b);
}
```

이렇게 할당을 하고 나면 입력을 받아야 겠지요? 위와 같은 `for` 문을 열심히 돌려서 입력을 받으면 됩니다.

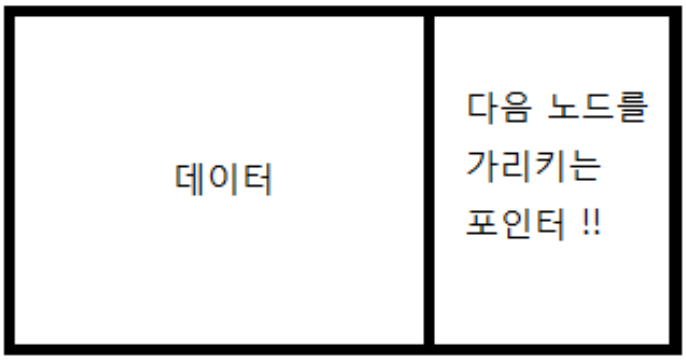
```
free(arr);
```

그리고 마지막에 위와 같이 `free` 로 깔끔하게 메모리를 정리해주는 것도 잊으면 안됩니다!

노드

여태까지 여러분들은 여러가지 자료형들을 배워왔습니다. 변수를 무식하게 나열하는 것을 막기 위해 배열을 이용하였고, 또 배열의 기능에 한계를 느낀 여러분은 구조체를 만들었습니다. 그리고 구조체 하나에 한 개 한 개를 다루는데 한계를 느낀 여러분은 구조체 배열을 이용해왔구요. 결국 배열로 다시 돌아왔습니다. 동적 할당을 함으로써 사용자가 원하는 크기의 입력을 다룰 수 있게 되었다고 하더라도 아직 많은 문제를 느끼고 있습니다. 만일 사용자가 마음이 변해서 한 개의 입력을 더 받고 싶다면 말이죠. 새롭게 동적할당을 하면 되지만 예컨대 1000 개 의 데이터가 있는데 1 개의 추가적인 데이터를 위해 1001 개를 위한 공간을 새로 잡으면 너무 아까운 것 같습니다.

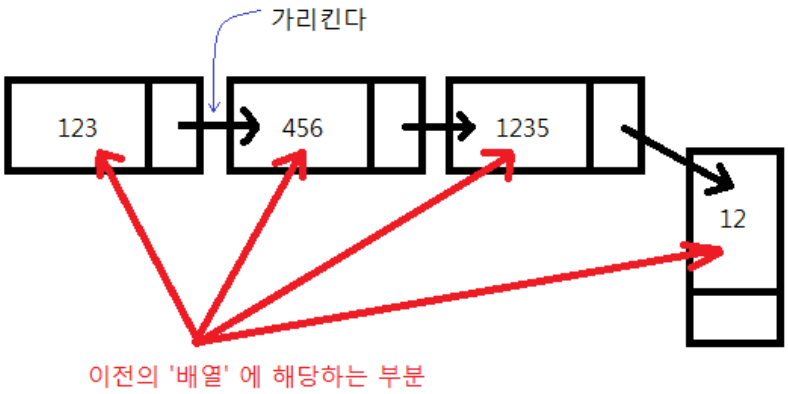
이를 해결하는 것이 바로 '노드' 입니다. 노드는 이렇게 생겼습니다.



상당히 단순하지요? 이를 C 코드로 나타내면 다음과 같습니다.

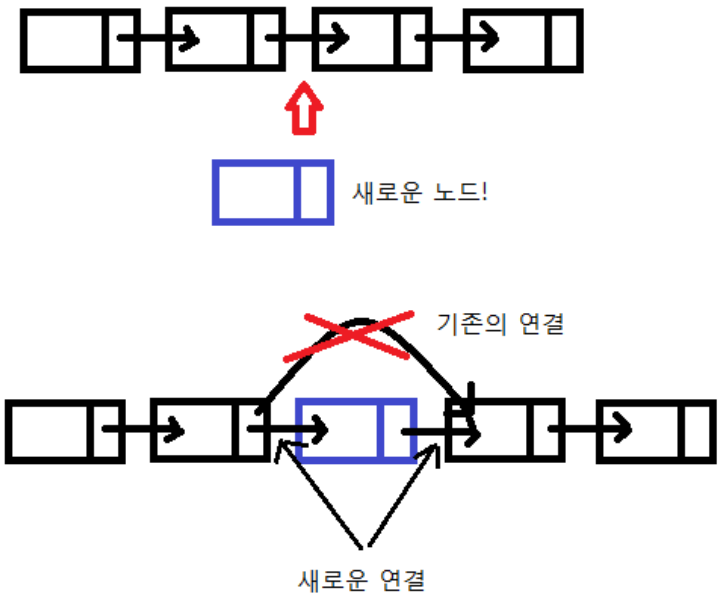
```
struct Node {
  int data;              /* 데이터 */
  struct Node* nextNode; /* 다음 노드를 가리키는 부분 */
};
```

아무튼 이렇게 생긴 노드를 어떻게 사용할까요?



위와 같이 사용합니다. 다시 말해 첫번째 노드가 다음 노드를 가리키면 다음 노드는 그 다음다음 노드를 가리키는 식으로 쭉 이어지며 마지막 노드 까지 이어지는데 마지막 노드는 아무것도 가리키지 않습니다. 또한 각각의 노드는 데이터를 하나씩 가지고 있지요. 다시 말해 나중에 데이터를 한 개 더 추가하려고 하면 마지막 노드에 새 노드를 만들어서 이어주기만 하면 됩니다.

뿐만 아닙니다. 기존의 배열에서는 거의 불가능 하였던 작업인 '배열 중간에 새 원소 집어넣기' 가 가능해집니다. 다시 말해 노드 사이에 새로운 노드를 끼워 넣을 수 있게 된다는 것이지요.



위 그림 처럼 기존에 있었던 연결을 없애버리고 그 사이에 새롭게 연결해주기만 하면 됩니다. 이러한 사실을 바탕으로 노드를 만들어봅시다. 가장 먼저 새로운 노드를 생성하는 `CreateNode` 함수

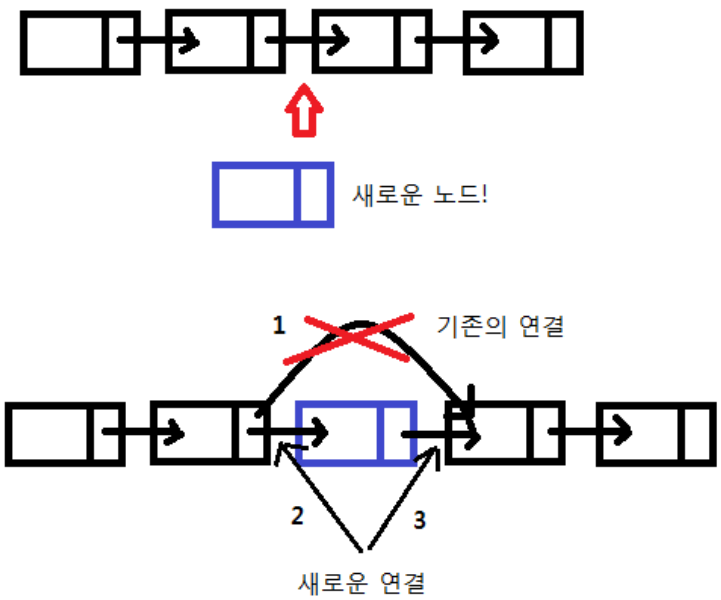
부터 만들어봅시다. 이 함수는 노드를 생성하기만 합니다. 노드를 생성하기 위해서는 데이터와 이 노드가 가리키는 다음 노드가 필요한데 이 함수는 단순히 첫번째 노드를 만드는 역할을 한다고 하고 nextNode 를 NULL 로 줍시다.

```
/* 새 노드를 만드는 함수 */
struct Node* CreateNode(int data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  newNode->data = data;
  newNode->nextNode = NULL;

  return newNode;
}
```

따라서 CreateNode 함수는 위와 같이 만들 수 있습니다. 일단 malloc 을 통해 노드를 메모리에 할당하였고 이 할당된 노드는 newNode 가 가리키게 됩니다. 이제, newNode->data 에 data 를 집어넣고 이 노드가 가리키는 다음 노드를 NULL 로 주면 됩니다.

사실 이 함수는 노드를 생성하기만 할 뿐 노드를 어떻게 관계짓지는 못합니다. 따라서 어떠한 노드 뒤에 새로운 노드를 생성하는 함수를 만들어야 할 것입니다. 이 함수는 InsertNode 함수라고 합시다. 따라서 어떠한 노드뒤에 올지 '앞에 있는 노드' 에 관한 정보와 '새로운 노드를 위한 데이터' 가 필요하므로 struct Node *current, int data 를 인자로 가져야 합니다.

```
/* current 라는 노드 뒤에 노드를 새로 만들어 넣는 함수 */
struct Node* InsertNode(struct Node* current, int data) {
  /* current 노드가 가리키고 있던 다음 노드가 after 이다 */
  struct Node* after = current->nextNode;

  /* 새로운 노드를 생성한다 */
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  /* 새 노드에 값을 넣어준다. */
  newNode->data = data;
  newNode->nextNode = after;

  /* current 는 이제 newNode 를 가리키게 된다 */
  current->nextNode = newNode;

  return newNode;
}
```



위 함수에 대한 설명을 위 그림을 보면서 해봅시다.

```
/* 새로운 노드를 생성한다 */
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

일단 위 문장을 통해 새로운 노드 newNode 를 생성하였습니다.

```
/* 새 노드에 값을 넣어준다. */
newNode->data = data;
newNode->nextNode = after;
```

이제 위 과정을 통해 newNode 에 nextNode 를 넣어주는 과정인데요, 이는 위 그림에서 3 번에 해당하는 과정입니다.

```
/* current 는 이제 newNode 를 가리키게 된다 */
current->nextNode = newNode;
```

또한 newNode 앞에 있던 노드의 nextNode 가 바뀌었으므로 새롭게 수정하는 과정이 위 코드인데 이는 위 그림에서 1,2 번에 해당하는 과정입니다. 이렇듯, 그림에 있던 과정들이 함수에 잘 구현이 되어 있음을 알 수 있습니다.

이렇게 노드를 잘 만들어주었다면 노드를 파괴하는 역할을 가지는 함수 역시 만들어야 합니다. 이를 위해서는 이 노드를 가리키고 있던 이전 노드가 필요하게 됩니다. 그런데 이 노드를 가리키고 있던

노드를 찾기 위해서는 맨 처음 부터 뒤져나가야 하는데, 맨 처음 노트를 헤드 라고 하며 우리의 DestoryNode 함수는 헤드를 인자로 받아야 합니다. 물론 파괴하고자 하는 노드도 인자로 받아야 하지요

```
/* 선택된 노드를 파괴하는 함수 */
void DestroyNode(struct Node *destroy, struct Node *head) {
  /* 다음 노드를 가리킬 포인터*/
  struct Node *next = head;

  /* head 를 파괴하려 한다면 */
  if (destroy == head) {
    free(destroy);
    return;
  }

  /* 만일 next 가 NULL 이면 종료 */
  while (next) {
    /* 만일 next 다음 노드가 destroy 라면 next 가 destory 앞 노드*/
    if (next->nextNode == destroy) {
      /* 따라서 next 의 다음 노드는 destory 가 아니라 destroy 의 다음 노드가
       * 된다. */
      next->nextNode = destroy->nextNode;
    }
    /* next 는 다음 노드를 가리킨다. */
    next = next->nextNode;
  }
  free(destroy);
}
```

위와 같이 만들면 됩니다. head 노드로 부터 차례 차례 하나 씩 다음 노드와 비교해가면서 찾아나가는 모습입니다. 이 과정은

```
while (next) {
  /* 만일 next 다음 노드가 destroy 라면 next 가 destory 앞 노드*/
  if (next->nextNode == destroy) {
    /* 따라서 next 의 다음 노드는 destory 가 아니라 destroy 의 다음 노드가 된다.
     */
    next->nextNode = destroy->nextNode;
  }
  /* next 는 다음 노드를 가리킨다. */
  next = next->nextNode;
}
```

에 잘 나타나 있습니다. 만일 next->nextNode == destroy 라면 next 의 다음 노드가 바로 destroy 가 되는 것이므로 거꾸로 생각해보면 destroy 를 가리키고 있었던 이전 노드는 next 가 됩니다. 이 때 destroy 는 메모리에서 파괴되어 사라지기 때문에 next 의 nextNode 가

destroy 가 되면 안되고 그 다음 다음 노드, 즉 destory 의 nextNode 가 되어야 한다는 것입니다. 따라서 위와 같은 과정을 수행하고침내 마지막에 free 를 해주면 됩니다.

이 과정을 한 소스에 정리하면

```c
#include <stdio.h>
#include <stdlib.h>
struct Node* InsertNode(struct Node* current, int data);
void DestroyNode(struct Node* destroy);
struct Node* CreateNode(int data);
void PrintNodeFrom(struct Node* from);

struct Node {
  int data;              /* 데이터 */
  struct Node* nextNode; /* 다음 노드를 가리키는 부분 */
};
int main() {
  struct Node* Node1 = CreateNode(100);
  struct Node* Node2 = InsertNode(Node1, 200);
  struct Node* Node3 = InsertNode(Node2, 300);
  /* Node 2 뒤에 Node4 넣기 */
  struct Node* Node4 = InsertNode(Node2, 400);

  PrintNodeFrom(Node1);
  return 0;
}
void PrintNodeFrom(struct Node* from) {
  /* from 이 NULL 일 때 까지,
     즉 끝 부분에 도달할 때 까지 출력 */
  while (from) {
    printf("노드의 데이터 : %d \n", from->data);
    from = from->nextNode;
  }
}
/* current 라는 노드 뒤에 노드를 새로 만들어 넣는 함수 */
struct Node* InsertNode(struct Node* current, int data) {
  /* current 노드가 가리키고 있던 다음 노드가 after 이다 */
  struct Node* after = current->nextNode;

  /* 새로운 노드를 생성한다 */
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  /* 새 노드에 값을 넣어준다. */
  newNode->data = data;
  newNode->nextNode = after;

  /* current 는 이제 newNode 를 가리키게 된다 */
  current->nextNode = newNode;

  return newNode;
} /* 선택된 노드를 파괴하는 함수 */
```

```
void DestroyNode(struct Node* destroy,
                 struct Node* head) { /* 다음 노드를 가리킬 포인터*/
  struct Node* next = head;           /* head 를 파괴하려 한다면 */
  if (destroy == head) {
    free(destroy);
    return;
  }              /* 만일 next 가 NULL 이면 종료 */
  while (next) { /* 만일 next 다음 노드가 destroy 라면 next 가 destory 앞 노드*/
    if (next->nextNode == destroy) { /* 따라서 next 의 다음 노드는 destory 가
                                        아니라 destroy 의 다음 노드가 된다. */
      next->nextNode = destroy->nextNode;
    } /* next 는 다음 노드를 가리킨다. */
    next = next->nextNode;
  }
  free(destroy);
}
/* 새 노드를 만드는 함수 */
struct Node* CreateNode(int data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  newNode->data = data;
  newNode->nextNode = NULL;

  return newNode;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
노드의 데이터 : 100
노드의 데이터 : 200
노드의 데이터 : 400
노드의 데이터 : 300
```

와 같이 잘 나옵니다.

```
void PrintNodeFrom(struct Node *from) {
  /* from 이 NULL 일 때 까지,
     즉 끝 부분에 도달할 때 까지 출력 */
  while (from) {
    printf("노드의 데이터 : %d \n", from->data);
    from = from->nextNode;
  }
}
```

일단 추가적으로 위와 같이 from 이후의 모든 노드의 값을 출력하는 함수인 PrintNodeFrom 이라는 함수를 정의하였습니다.

```
struct Node* Node1 = CreateNode(100);
struct Node* Node2 = InsertNode(Node1, 200);
struct Node* Node3 = InsertNode(Node2, 300);
/* Node 2 뒤에 Node4 넣기 */
struct Node* Node4 = InsertNode(Node2, 400);
```

메인 함수에서 위와 같이 Node 들을 정의하였습니다. 먼저 헤드 노드인 Node1 을 CreateNode 함수를 통해 정의하였고, Node1 뒤에 Node2 , Node2 뒤에 Node3, 그리고 Node2 뒤에 Node4 를 끼워넣었습니다. 그리고 실행한 결과 100, 200, 400, 300 순으로 제대로 나온 것을 볼 수 있습니다.

이렇게 노드는 배열과는 달리 추가/삭제/삽입이 월등히 편리합니다. 그렇다고 해서 노드가 배열보다 월등한 것일까요? 꼭 그렇다고는 말할 수 없습니다. 왜냐하면 배열의 경우 3 번째 원소에 접근하기 위해서는 단순히 arr[3] 으로 하면 되지만 노드의 경우 헤드로 부터 3 번째 까지 일일히 찾아가야만 하기 때문이죠. 따라서 N 개의 노드가 있다면 최악의 경우 N 번동안 계속 찾아야 하지만 배열의 경우 특정한 상수 시간 내에 찾아갈 수 있기 때문에 이부분에서는 배열이 월등히 좋다고 할 수 있습니다. 또한 노드의 경우 데이터를 위한 공간 말고도 다음 노드를 가리키기 위한 4 바이트가 더 필요하기 때문에 공간적으로도 약간 손해를 본다고 생각할 수 있습니다.

따라서 결론적으로 이야기 하자면 추가/삭제/삽입이 자주 일어나는 경우 노드를 사용하고 특정한 번째에 찾아가야 하는 일이 잦은 일은 배열을 사용하는 것이 이롭다는 것을 알 수 있습니다.

사실 노드 말고도 여러가지 형태의 자료 구조들이 있는데 예를 들면 스택, 큐, 트리 등이 있다고 볼 수 있습니다. 이들에 관한 자세한 내용은 여러분 스스로 찾아 보시기 바랍니다.

메모리 관련 함수

이번 단원이 메모리에 관한 것인 만큼 메모리에 관해서는 빠삭하게 알아가도록 합시다. 이를 위해 메모리에 관련된 C 표준 라이브러리에서 기본으로 지원되는 것들에 대해 알아보도록 합시다. 일단 메모리를 직접적으로 가지고 논다고 말할 수 있는 함수들은 memmove, memcpy, memcmp, memset 등이 있는데 우리는 여기서 대표적인 3 개의 함수인 memmove, memcpy, memcmp 만 알아보도록 합시다. 이 함수들 모두 string.h 에 정의되어 있습니다.

먼저 memcpy 함수 부터 봅시다.

```
/* memcpy 함수 */

#include <stdio.h>
#include <string.h>
```

```c
int main() {
  char str[50] = "I love Chewing C hahaha";
  char str2[50];
  char str3[50];

  memcpy(str2, str, strlen(str) + 1);
  memcpy(str3, "hello", 6);

  printf("%s \n", str);
  printf("%s \n", str2);
  printf("%s \n", str3);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
I love Chewing C hahaha
I love Chewing C hahaha
hello
```

와 같이 나옵니다.

memcpy 함수는 메모리의 특정한 부분으로 부터 얼마 까지의 부분을 다른 메모리 영역으로 복사해 주는 함수 입니다. 위와 같이 문자열을 복사하는데 사용될 수 있죠. 물론 문자열 복사를 전문적으로 하는 함수는 strcpy 이지만 위와 같이 memcpy 함수를 사용하는 것도 나쁘지 않습니다.

```c
memcpy(str2, str, strlen(str) + 1);
```

일단 위 문장은 'str 로 부터 strlen(str) + 1 만큼의 문자를 str2 로 복사해라' 라는 의미입니다. 이 때, strlen 함수는 문자열의 길이를 리턴해주는 함수로 예를 들어 strlen("abc"); 를 하면 3 이 리턴됩니다. 이 때 마지막의 NULL 문자는 세지 않으므로 str2 에 memcpy 로 복사할 때 에는 1 을 더한만큼을 더 복사해주어야 합니다.

```c
memcpy(str3, "hello", 6)
```

마찬가지로 str3 의 경우도 hello 의 5 문자와 끝에 NULL 을 위해 총 6 문자를 hello 의 시작 주소로 부터 복사를 하게 됩니다. memcpy 에 관한 자세한 내용은 여기 을 참조하시기 바랍니다.

다음으로 memmove 함수에 대해 살펴봅시다. 이 함수는 메모리의 특정한 부분의 내용을 다른 부분으로 옮겨주는 역할을 합니다. 이 때 '옮긴다' 고 해서 이전 공간에 있던 데이터가 사라지지는 않습니다.

```c
/* memmove 함수 */

#include <stdio.h>
#include <string.h>

int main() {
  char str[50] = "I love Chewing C hahaha";

  printf("%s \n", str);
  printf("memmove 이후 \n");
  memmove(str + 23, str + 17, 6);
  printf("%s", str);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
I love Chewing C hahaha
memmove 이후
I love Chewing C hahahahahaha
```

와 같이 나옵니다.

```c
char str[50] = "I love Chewing C hahaha";
memmove(str + 23, str + 17, 6);
```

memmove 함수는 위 경우 str+17 에서 6 개의 문자를 str+23 에 옮겼습니다. 다시 말해 hahaha 의 시작 부분에서 6 개의 문자인 "hahaha" 를 str 의 맨 마지막 부분으로 복사해 넣었다는 뜻입니다. 다시 말해 str 뒤에 "hahaha" 를 추가하게 된 셈이지요. 이를 통해 문자열을 *I love Chewing C hahahahahaha* 로 만들 수 있게 되었습니다. memmove 함수의 장점은 memcpy 와 하는 일이 많이 비슷해보이지만 사실 memcpy 와는 달리 메모리 공간이 겹쳐도 됩니다. 위 경우도 str 과 복사하는 부분이 겹쳤지만 성공적으로 복사가 수행되었습니다. 덕분에 나중에는 memmove 함수를 아주 많이 사용하게 될 것입니다.

마지막으로 memcmp 함수를 살펴보도록 합시다. 이는 이름에서도 충분히 짐작이 되듯이 두 개의 메모리 공간을 서로 비교하는 함수 입니다.

```c
/* memcmp 함수 */

#include <stdio.h>
#include <string.h>

int main() {
  int arr[10] = {1, 2, 3, 4, 5};
  int arr2[10] = {1, 2, 3, 4, 5};

  if (memcmp(arr, arr2, 5) == 0)
    printf("arr 과 arr2 는 일치! \n");
  else
    printf("arr 과 arr2 는 일치 안함 \n");

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
arr 과 arr2 는 일치!
```

와 같이 나옵니다.

memcmp 함수는 꽤 유용하게 사용될 수 있습니다. 이 함수는 메모리의 두 부분을 원하는 만큼 비교를 합니다. 이 때 같다면 0, 다르다면 결과에 따라 0 이 아닌 값을 리턴하게 되지요.

```c
if (memcmp(arr, arr2, 5) == 0)
```

위 문장의 경우 arr 과 arr2 를 비교해서 처음 5 개의 바이트가 같다면 0 을 리턴하게 됩니다. 주의해야 할 점은 '5 개의 원소' 가 아니라 5 바이트 라는 점 이지요. 만일 arr1 과 arr2 전체를 비교하고 싶다면 3 번째 인자로 sizeof(int) * 5 를 넣어 주어야 했었을 것입니다.

이렇게 메모리를 가지고 노는 3 개의 함수들을 모두 살펴보았습니다. 저는 이 함수를 사용하는 아주 기본적인 방법만을 가르쳐 주었을 뿐 이 함수들을 어떻게 응용시켜서 적용시키냐는 여러분들의 몫입니다. 이제, 메모리를 아주 빠삭하게 다룰 수 있기 되었으니 (동적 메모리 할당도 할 줄 알고, memmove 와 같은 놀라운 메모리 관련 함수들도 사용할 줄 아니..) 이번 강좌는 여기서 끝마치도록 하겠습니다.

생각 해보기

앞서 배운 노드는 여러모로 생각해볼 점이 많다. 다음의 과제들을 차례대로 해결해보기 바랍니다.

문제 1

head 가 주어질 때 전체 노드의 개수를 세는 `int CountNode(Node* head)` 함수를 작성하시오 (난이도 : 下)

문제 2

head 와 원하는 노드가 주어질 때 원하는 노드의 데이터 값을 출력하는 `int SearchNode(Node* head, Node *search)` 함수를 작성하시오 (난이도 : 下)

문제 3

앞서 구현하였던 Node 의 단점으로 '이 노드를 가리키는 노드' 를 쉽게 알 수 없다는 점이다. 이를 보완하기 위해

```
struct Node {
  int data;               /* 데이터 */
  struct Node* nextNode; /* 다음 노드를 가리키는 부분 */
  struct Node* prevNode; /* 이전 노드를 가리키는 부분 */
};
```

형식으로 노드를 만들어보고 앞서 작성했던 모든 함수들을 다시 작성해보시오 (난이도 : 中)

문제 4

위와 같은 형식의 노드를 개량하여 head 가 맨 마지막 노드인 tail 을 prevNode 로 가리키는 원형의 노드를 만들어보시오. 다시 말해 노드의 처음과 끝이 없다고 볼 수 있다. 이러한 형태의 노드를 이용하여 앞서 구현하였던 모든 함수를 구현해보시오 (난이도 : 中上)

문제 5

이전 강좌에서 만들었던 도서 관리 프로그램을 동적 할당과 구조체를 이용하여 만들어보세요 (난이도 : 中)

19

매크로 함수, 인라인 함수

안녕하세요 여러분. 저의 36 번째 강좌가 시작되었습니다. 요즘에 뒤로 갈 수 록 강좌들의 댓글 수가 적어지는 것을 보아 처음에 큰마음 먹고 강좌 보기를 시작하였다가 시간적 여유의 한계나 온라인 상의 한계를 느끼고 포기하신 분들이 많은 것 같은데 과연 누가 여기 까지 성공적으로 달려왔는지 궁금하네요. 전체 강좌의 앞부분은 C 언어 자체를 아는데 주력하였다면 후반으로 갈 수록 C 언어 자체를 이해하기 보다는 C 언어와 친해지는 과정으로 진행이 되고 있습니다.

아무튼. 적어도 제 강좌만 다 이해한다면 C 의 기초 부분은 훌륭하게 다진 프로그래머로 만들어 줄 수 있으니 열심히 따라와주세요:)

매크로 함수

```c
/* 매크로 함수*/
#include <stdio.h>
#define square(x) x *x

int main(int argc, char **argv) {
  printf("square(3) : %d \n", square(3));

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
square(3) : 9
```

와 같이 나옵니다.

매크로 함수를 정의하는 방법은 아래와 같습니다.

#define 함수 이름(인자) 치환할 것

그렇다면

```
#define square(x) x* x
```

위 문장의 의미는 square 라는 이름의 매크로 함수고 인자로 x 를 x*x 로 치환한다는 의미 입니다. #define 문은 앞서 배웠듯이 '어떠한 것을 다른 것으로 치환해주는 것' 이라고 배웠습니다. 여기서도 그 역할이 똑같습니다. squre(x) 부분을 x*x 로 치환해주게 되지요. 따라서

```
printf("square(3) : %d \n", square(3));
```

위 문장은

```
printf("square(3) : %d \n", 3 * 3);
```

과 정확히 동일하게 됩니다. 위와 같은 것을 '매크로 함수' 라 부르는 이유는 정말 하는 일이 함수와 비슷하기 때문입니다. 만일 우리가 int squre(x) 라는 함수를 만들어서 x*x 를 리턴하게 하였다면 말그대로 3*3 이 리턴될 것이지요. 하지만 이 매크로 함수와 진짜 함수는 엄연한 차이가 있습니다.

```
printf("square(3) : %d \n", square(3));
```

우리가 컴파일러라고 생각하고 위 문장을 어떻게 해석할지 봅시다. 만일 int squre(int x) 라는 실제 함수가 있다면 'square 라는 함수를 호출해서 인자에 3 을 전달하고 9 를 리턴한다' 가 됩니다. 하지만 매크로 함수는 위 문장이 컴파일 되기 전에 전처리기에 의해 그냥

```
printf("square(3) : %d \n", 3 * 3);
```

로 바뀌어 버립니다. 여기서 중요한 점은 '컴파일 되기 전에' 부분에 있지요. 다시 말해 컴퓨터는 함수를 호출하고 뭐시기 뭐시기 하는 부분 없이 그냥 3*3 을 계산해 버립니다. 이는 앞서 보았던 #define 문을 통한 치환과 동일합니다.

```
/* 매크로 함수*/
#include <stdio.h>
#define square(x) x *x

int main(int argc, char **argv) {
```

```
  printf("square(3) : %d \n", square(3 + 1));

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
square(3) : 7
```

흠. 조금 놀라운 결과가 나왔습니다.

분명히

```
printf("square(3) : %d \n", square(3 + 1));
```

위 문장에서 우리의 의도는 4*4 를 계산하는 것이였을 것입니다. 그런데 16 이 아니라 전혀 엉뚱한 답인 7 이 나왔습니다. 도대체 왜 이런 일이 벌어진 것일까요? 제가 아까 한 말, 매크로 함수는 단순히 '싹 치환해 버리는 것이다' 라는 것을 상기 시켜 보세요. 전처리기에서 square(3+1) 이 어떻게 바뀔지 말이지요. 단순히 생각해보면

```
printf("square(3) : %d \n", 3 + 1 * 3 + 1);
```

이 됩니다. 그런데 놀랍게도 맞습니다. 컴퓨터는 실제로 위 문장을 위와 같이 치환해버립니다. 따라서 결과적으로 7 이 출력된 것이지요. 이러한 문제를 해결하기 위해서는 어떻게 해야 할까요. 방법은 단순합니다.

```
#define square(x) x* x
```

를

```
#define square(x) (x) * (x)
```

로 바꾸어 주기만 하면 됩니다. 그렇게 된다면 printf 안에 있었던 문장은

```
printf("square(3) : %d \n", (3 + 1) * (3 + 1));
```

가 되어 16 이 성공적으로 출력될 것입니다.

```c
/* 라디안에서 도로 바꾸기*/
#include <stdio.h>
#define RADTODEG(x) (x) * 57.295

int main(int argc, char **argv) {
  printf("5 rad 는 : %f 도", RADTODEG(5));

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
5 rad 는 : 286.475000 도
```

아마 위 소스 코드의 내용 보다 라디안(radian) 이 무엇인지 모르는 분들이 더 많을 듯 한데, 그냥 말하자면 원주의 길이와 반지름의 길이의 비를 이용해 각도를 나타내는 단위로 자세한 내용은 생략하고 여기서는 단순히 `1 rad` 는 `57.295` 도 라는 것만 아시면 됩니다.

```c
#define RADTODEG(x) (x) * 57.295
```

위와 같이 라디안에서 각도로 변환하는 매크로 함수를 정의하였습니다. 그리고 `printf` 에서

```c
printf("5 rad 는 : %f 도", RADTODEG(5));
```

와 같이 매크로 함수를 이용하였을 때

```c
printf("5 rad 는 : %f 도", 5 * 57.295);
```

로 잘 바뀌어 값이 잘 출력되었습니다. 물론 `RADTODEG(1+4)` 를 하더라도 `(1+4) * 57.295` 로 잘 바뀌어 원하는 결과를 출력할 수 있게 되지요. 그렇다면 위와 같이 정의한 `RADTODEG` 는 문제가 없을까요?

한 5 초 만 생각해보세요. 위와 같이 완벽하게 잘 정의했다고 하더라도 문제는 있습니다. 바로 아래와 같은 문장이지요

```c
printf("5 rad 는 : %f 도", 1 / RADTODEG(5));
```

위 문장은 전처리기에 의해

```
printf("5 rad 는 : %f 도", 1 / 5 * 57.295);
```

로 바뀝니다. 이는 우리가 원하는 결과인 `1 / (5 * 57.295)` 와 전혀 다른 것이지요. 이와 같은 문제를 막기 위해서 역시 전체 수식을 소괄호로 감싸 `#define RADTODEG(x) ((x) * 57.295)`와 같이 만들어야 합니다. 상당히 귀찮은 일이지요. 사소한 실수 하나로 꽤 큰 문제가 초래될 수 있으니까요.

```
/* 변수의 이름 출력하기 */
#include <stdio.h>
#define PrintVariableName(var) printf(#var "\n");

int main(int argc, char **argv) {
  int a;

  PrintVariableName(a);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
a
```

와 같이 잘 나옵니다.

```
#define PrintVariableName(var) printf(#var "\n");
```

위 문장의 의미 부터 살펴봅시다. `#define` 과 같은 전처리기 문에서만 사용되는 것 중 # 이 있는데, 어떠한 인자 앞에 # 을 붙이게 되면 이 인자를 문자열로 바꾸어 버립니다. 따라서

```
PrintVariableName(a);
```

은

```
printf(
  "a"
  "\n");
```

으로 바뀌게 되지요. 이 때 C 언어에서 연속한 두 개의 문자열은 그냥 하나로 합쳐지므로 위 코드는 그냥

```
printf("a\n");
```

와 동일하게 됩니다. 따라서 결과적으로 a 가 화면에 출력되는 것이지요. 여러분은 # 가 붙으면 단순히 '이 것을 문자열로 바꾼다' 라고 생각만 해주시면 됩니다.

```
/* ## 의 사용 */
#include <stdio.h>
#define AddName(x, y) x##y

int main(int argc, char **argv) {
  int AddName(a, b);

  ab = 3;

  printf("%d \n", ab);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
3
```

와 같이 나옵니다.

이번에는 # 의 친구 격인 ## 에 대해 보도록 합시다. 한 가지 명심할 점은 # 나 ## 모두 '전처리기 문' 에서만 사용할 수 있다는 것입니다. 즉 #define 에서만 사용할 수 있다는 정도로만 알아두세요. ## 문은 아마 짐작했지만 입력된 것을 하나로 '합쳐주는' 역할을 합니다.

```
#define AddName(x, y) x##y
```

위와 같이 AddName 에서는 x##y 는 x 에 있는 것과 y 에 있는 것을 하나로 합쳐줍니다. 따라서

```
int AddName(a, b);
```

이 부분은 전처리기에 의해

```
int ab;
```

로 정확히 치환됩니다. 따라서 컴파일러는 ab 라는 이름의 변수를 선언하게 되고 그 뒤로 쭉 가는 것이지요.

매크로함수가 위와 같이 여러 편리한 점들은 있지만 앞서 집고 나간 것 처럼 여러가지 어려운 문제점들도 많습니다. 위에서 처럼 괄호를 제대로 쓰지 않아 오류가 나는 경우가 많은데 이 경우 디버깅하기가 매우 까다롭기 때문에 오랜 시간을 잡아먹는 경우도 많습니다. 이러한 문제를 해결하기 위해 C 언어에서는 또다른 해결책을 제시하였는데요, 이는 바로 **인라인(inline)** 함수 입니다.

인라인 함수

```c
/* 인라인 함수 */

#include <stdio.h>
__inline int square(int a) { return a * a; }
int main(int argc, char **argv) {
  printf("%d", square(3));

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
9
```

음. 일단 내용만을 보아서는 크게 어렵지 않습니다.

```c
__inline int square(int a) { return a * a; }
```

위 부분은 `__inline` 을 빼고 본다면 단순히 `square` 라는 함수를 만든데 지나지 않습니다. 또한 `printf` 에서도 역시 함수를 호출했던 것처럼 똑같은 방식으로

```c
printf("%d", square(3));
```

로 사용하고 있지요. 하지만 이는 함수와는 전혀 다른 행동을 합니다. 함수의 경우 호출을 하게 되면 프로그램의 흐름이 완전히 다른 곳으로 넘어가게 됩니다. 예를 들어서

```
int cubic(int a) { return a * a * a; }
```

와 같은 세제곱을 하는 '평범한' 함수 하나를 만들고, main 함수에서

```
int main(int argc, char **argv) {
  printf("%d", cubic(3));
  return 0;
}
```

와 같이 cubic 함수를 호출을 하게 된다면, cubic (3) 을 실행 시, 프로그램의 흐름이 main 함수를 벗어나 메모리 어딘가에 위치한 cubic 함수에 찾아가서 인자로 3 을 전달하고 27 을 리턴하는데, 그 리턴값을 가지고 다시 main 함수로 돌아오게 되는 것이지요. 이렇게 함수를 사용하게 되면 프로그램의 흐름이 기존의 함수 내부에서 벗어나 다른 함수에 들렸다가 오게 되는데 이러한 과정을 줄여서 '함수를 호출하는 과정' 이라고 말하게 됩니다.

이렇게 함수를 호출하게 된다면 단순히 a 를 세 번 곱하는 작업인데도 시간이 꽤 걸리게 되죠. 즉 cubic 함수처럼 단순한 작업만을 하는 함수의 경우에는 굳이 함수로 따로 만들 필요 없이 차라리 main 함수 내에서

```
printf("%d", 3 * 3 * 3);
```

으로 하는 것이 훨씬 효율적일 것입니다.

이러한 생각을 살려 만든 것이 inline 함수 입니다. 위에서 inline 형식으로 만든 square 함수는 우리가 생각하는 함수가 전혀 아닙니다. 단순히 '함수 처럼 보이는 것' 일 뿐이지요. inline 함수를 사용하게 되면 마치 매크로 함수처럼

```
int main(int argc, char **argv) {
  printf("%d", square(3));
  return 0;
}
```

위 코드가

```
int main(int argc, char **argv) {
  printf("%d", 3 * 3);
  return 0;
}
```

과 정확히 동일해집니다. 한 가지 매크로 함수와 차이점이 있다면 매크로 함수와는 달리 인라인 함수는 전처리기가 무식하게 치환해 버리는 것이 아닙니다. 매크로 함수를 사용 했었을 때 에는 전처리기가 무식하게 치환해 버리는 바람에 연산자 우선 순위를 정확하게 고려해서 괄호도 적당히 묶어주고 해야겠지만 인라인 함수의 경우 똑똑한 컴파일러가 인라인 함수를 사용한 문장 내부에서 적절하게 '우리가 보통 함수를 사용하는 것처럼' 바꿔 줍니다.

다시 말해

```
int main(int argc, char **argv) {
  printf("%d", square(3 + 1));
  return 0;
}
```

과 같은 문장은 우리가 보통 함수를 사용 하는 것처럼 처리가 되기 때문에 똑똑하게

```
int main(int argc, char **argv) {
  printf("%d", (3 + 1) * (3 + 1));
  return 0;
}
```

이 됩니다.

```
/* 다른 인라인 함수 예제*/
#include <stdio.h>
__inline int max(int a, int b) {
  if (a > b)
    return a;
  else
    return b;
}
int main(int argc, char **argv) {
  printf("3 과 2 중 최대값은 : %d", max(3, 2));
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
3 과 2 중 최대값은 : 3
```

와 같이 나옵니다.

이번에도 역시 인라인 함수를 사용하였습니다. 컴파일러는 max(3,2) 라는 문장을 보고 max 함수 내부의 코드로 max(3,2) 를 대체할 수 있을 지 생각하게 됩니다.

만일 이를 max 함수를 호출하고 리턴 받는 형태 보다, 직접 작업하는 비용이 더 작다고 생각할 때에는 이를 치환하게 됩니다. 하지만, 이 비용이 더 크다면 (코드 길이가 더 길어지겠지요?) inline 키워드를 무시하게 됩니다.

다행히도 이 max 함수의 경우 간단히 치환할 수 있는데 있는데 아마 컴파일러는 이 함수의 내용을

```
int main(int argc, char **argv) {
  printf("%d 과 %d 중 최대값은 : %d \n", 3, 2, 3 > 2 ? 3 : 2);
  return 0;
}
```

위와 같은 문장으로 변환 시켜서 3 이 출력되게 할 것입니다.[1)]

여태까지 인라인 함수와 매크로 함수를 살펴 보았는데, 많은 사람들은 매크로 함수 보다는 인라인 함수를 사용하도록 권장하고 있습니다. 이 때문에 원래는 C 표준에 없던 인라인 함수가 C99 라는 새로운 C 표준에 포함된 것만 보아도 알 수 있습니다.

인라인 함수는 매크로 함수와는 달리 **컴파일러**가 처리하기 때문에 훨씬 더 똑똑하게 동작하는데 일단, 매크로 함수와는 달리 인라인 함수는 인자들의 타입을 확인합니다.

또한 인라인 함수는 매크로와는 달리 단순 치환을 하는 것이 아니라 진짜 함수 처럼 동작하기 때문에 훨씬 구현하기 쉽고 편리합니다. 뿐만 아니라 디버깅 역시 인라인 함수가 편리하지요.

앞으로 여러분들은 단순한 작업들을 보기 편하게 함수로 처리하고 싶을 때 에는 인라인 함수들을 적극적으로 사용하였으면 합니다 :)

생각 해 보기

문제 1

다음과 같은 인라인 함수를 하나의 문장으로 바꿀 수 있는지 생각해보세요.

```
__inline int some_function(int a) {
  if (a == 0)
    return 1;
  else if (a == 1)
    return 3;
  else
```

1) 컴파일러가 컴파일을 수행할 때 실제로 저 코드로 바꿔서 컴파일 하지는 않습니다. 코드가 컴파일 된 결과가 위 코드가 컴파일 된 결과와 같다는 뜻입니다.

```
    return a * 2;
}
```

20

그 밖에 키워드들

안녕하세요 여러분. 저의 C 언어 강의도 이제 막바지에 다다랐습니다. 정말로 첫번째 강의부터 여기까지 달려오셨다면 정말 대단하다고 말씀 드리고 싶네요. 마라톤에 비유하자면, 42.195km 에서 한 40km 정도 까지 열심히 뛰어 왔다고 보시면 됩니다. 그럼, 나머지 2.195km 도 더 뛸 의향이 있겠죠?

```c
/* 루저 위너 판별*/
#include <stdio.h>
int Print_Status(struct HUMAN human);
struct HUMAN {
  int age;
  int height;
  int weight;
  int gender;
};

int main() {
  struct HUMAN Adam = {31, 182, 75, 0};
  struct HUMAN Eve = {27, 166, 48, 1};

  Print_Status(Adam);
  Print_Status(Eve);
}

int Print_Status(struct HUMAN human) {
  if (human.gender == 0) {
    printf("MALE \n");
  } else {
    printf("FEMALE \n");
  }

  printf("AGE : %d / Height : %d / Weight : %d \n", human.age, human.height,
         human.weight);

  if (human.gender == 0 && human.height >= 180) {
    printf("HE IS A WINNER!! \n");
```

```
  } else if (human.gender == 0 && human.height < 180) {
    printf("HE IS A LOSER!! \n");
  }

  printf("---------------------------------------- \n");

  return 0;
}
```

위 코드는 이전에 구조체 단원에서 만들었던 루저-위너 구별 프로그램 입니다. 그런데 위 코드에서 한 가지 귀찮은 점이 있습니다. 바로, 구조체를 사용할 때 마다 앞에 `struct` 키워드를 붙여야 한다는 점입니다. 이게 상당히 짜증나는 일인데, 간혹 `struct` 키워드를 붙이지 않는 날에는

컴파일 오류

```
error C2146: 구문 오류 : ')'이(가) 'human' 식별자 앞에 없습니다.
error C2061: 구문 오류 : 식별자 'human'
error C2059: 구문 오류 : ';'
error C2059: 구문 오류 : ')'
error C2449: 파일 범위에 '{'가 있습니다. 함수 헤더가 없는 것 같습니다.
error C2059: 구문 오류 : '}'
```

위와 같은 오류 테러를 맞보게 됩니다. (위 오류는 `int Print_Status(struct HUMAN human)` 대신에 `int Print_Status(HUMAN human)` 이라 썼을 때 나타나는 오류들 입니다) 상당히 짜증일입니다. 그렇다면 매번 귀찮게 `struct HUMAN` 이라 쓰는 대신에 간단하게 쓰는 방법이 없을까요?

물론 있습니다.

```
/* typedef 의 이용 */
#include <stdio.h>
struct HUMAN {
  int age;
  int height;
  int weight;
  int gender;
};

typedef struct HUMAN Human;
int Print_Status(Human human);
int main() {
  Human Adam = {31, 182, 75, 0};
  Human Eve = {27, 166, 48, 1};

  Print_Status(Adam);
```

```
  Print_Status(Eve);
}

int Print_Status(Human human) {
  if (human.gender == 0) {
    printf("MALE \n");
  } else {
    printf("FEMALE \n");
  }

  printf("AGE : %d / Height : %d / Weight : %d \n", human.age, human.height,
         human.weight);

  if (human.gender == 0 && human.height >= 180) {
    printf("HE IS A WINNER!! \n");
  } else if (human.gender == 0 && human.height < 180) {
    printf("HE IS A LOSER!! \n");
  }

  printf("------------------------------------------ \n");

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
MALE
AGE : 31 / Height : 182 / Weight : 75
HE IS A WINNER!!
------------------------------------------
FEMALE
AGE : 27 / Height : 166 / Weight : 48
------------------------------------------
```

와 같이 나옵니다.

위 코드에서 가장 눈여겨 보아야 할 부분은

```
typedef struct HUMAN Human;
```

입니다. 우리는 여기서 typedef 라는 키워드를 사용했는데 이 키워드는 다음과 같이 사용합니다

typedef (이름을 새로 부여하고자 하는 타입) (새로 준 타입의 이름)

다시 말해 위에서 썼던 코드는 `struct HUMAN` 이라는 타입에 `Human` 이라는 다른 이름을 붙인 것이 됩니다. 즉, `struct HUMAN` 이라고 쓸 것을 `Human` 이라고 써도 된다는 것이지요. 물론 기존의 이름을 없애 버린 것이 아니기 때문에 `typedef` 를 사용한 이후에도 `struct HUMAN` 이라고 쓴 것은 유효 합니다.

즉 위와 같은 일을 하고 나면 다음과 같은 문장은 모두 동일해집니다.

```
struct HUMAN a;
Human a;
```

상당히 편리해졌지요? 하지만 진정 `typedef` 를 이용하는 이유는 이렇게 형을 간단하게 쓴다는 이유 때만은 아닙니다. 아래의 예제를 보세요.

```
/* 간단한 계산기 프로그램 */
#include <stdio.h>
int main() {
  int input;
  int a, b;

  while (1) {
    printf("--- 계산기 --- \n");
    printf("1. 덧셈 \n");
    printf("2. 뺄셈 \n");
    printf("3. 종료 \n");

    scanf("%d", &input);

    if (input == 1) {
      printf("두 수 : ");
      scanf("%d%d", &a, &b);
      printf("%d 와 %d 의 합 : %d \n", a, b, a + b);
    } else if (input == 2) {
      printf("두 수 : ");
      scanf("%d%d", &a, &b);
      printf("%d 와 %d 의 차 : %d \n", a, b, a - b);
    } else
      break;
  }

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
--- 계산기 ---
1. 덧셈
2. 뺄셈
3. 종료
1
두 수 : 123
124
123 와 124 의 합 : 247
--- 계산기 ---
```

와 같이 계산기 프로그램이 잘 실행됩니다. 만일 이 프로그램을 실제 계산기에 사용한다고 합시다. 그런데, 우리가 이 프로그램을 사용할 계산기는 안타깝게도 CPU 에서 32 비트 정수의 연산을 할 수 없습니다. 오직 16 비트 이하만 연산할 수 있다고 합시다. 그렇다면 이 계산기에서 int 형을 사용하는 것은 불가능 하고 short 나 char 형의 변수들만 선언해야 겠죠.

그렇다면 이를 위해 소스 코드 전체의 모든 변수들을 char 이나 short 로 바꿔주어야 합니다. 그런데 만일 동일한 프로그램인데 다른 기종의 계산기 에서는 int 형이 사용 가능하다고 합시다. 그렇다면 이 코드를 다시 또 바꿔주어야 합니다. 아주 아주 귀찮은 일이 아닐 수 없죠. 이런 상황을 대비하여서 다음과 같이 코드를 바꿔봅시다.

```c
/* 향상된 소스 코드 */
#include <stdio.h>
typedef int CAL_TYPE;
int main() {
  CAL_TYPE input;
  CAL_TYPE a, b;

  while (1) {
    printf("--- 계산기 --- \n");
    printf("1. 덧셈 \n");
    printf("2. 뺄셈 \n");
    printf("3. 종료 \n");

    scanf("%d", &input);

    if (input == 1) {
      printf("두 수 : ");
      scanf("%d%d", &a, &b);
      printf("%d 와 %d 의 합 : %d \n", a, b, a + b);
    } else if (input == 2) {
      printf("두 수 : ");
      scanf("%d%d", &a, &b);
      printf("%d 와 %d 의 차 : %d \n", a, b, a - b);
```

```
    } else
      break;
  }

  return 0;
}
```

이 역시 잘 실행됩니다. 다만 바뀐 것은 변수들의 타입을 CAL_TYPE 라고 했던 점이지요. 그리고 위에서 typedef 를 통해 CAL_TYPE 가 int 형과 같다고 정의하였습니다. 만일 이 소스 코드를 short 나 char 만 되는 계산기에 적용시킬려면 어떻게 해야 할까요? 기존에는 모든 변수의 타입을 전부다 수정해야 했지만 이제는 typedef 에서 CAL_TYPE 의 형을 short 나 char 로 간단히 바꿔버리면 되는 것입니다. 정말로 일이 쉬워졌습니다.

여러가지 typedef 들

```c
/* 여러가지 typedef 예제들 */

#include <stdio.h>
int add(int a, int b) { return a + b; }
typedef int CAL_TYPE;
typedef int (*Padd)(int, int);
typedef int Arrays[10];
int main() {
  CAL_TYPE a = 10;
  Arrays arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  Padd ptr = add;
  printf("a : %d \n", a);
  printf("arr[3] : %d \n", arr[3]);
  printf("add(3, 5) : %d \n", ptr(3, 5));
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
a : 10
arr[3] : 4
add(3, 5) : 8
```

가장 먼저 소스 코드에서 아래 문장 부터 살펴봅시다.

```
typedef int (*Padd)(int, int);
```

이전에 배웠던 함수 포인터가 잘 기억이 나시는지는 잘 모르겠지만 아무튼, 위 typedef 명령문은 복잡한 함수 포인터 명령을 Padd 라는 이름을 붙이는 것입니다. 즉, 다음 문장은 정확히 동일해집니다.

```
int (*ptr)(int, int) = add;
Padd ptr = add;
```

참고로 간혹

```
typedef int (*Padd)(int, int);
```

문장을 잘못 이해 하셔서, "우리가 앞에서 배운 바에 따르면 위 문장은 int 라는 형에 (*Padd)(int, int) 라는 또다른 이름을 붙이는 것이 아닌가?" 라고 물으실 수 있는데, 그건 아니고 Padd 라는 이름을 붙여주신다고 생각하시면 편합니다. 마찬가지로

```
typedef int Arrays[10];
```

도 역시 '원소가 10 개인 int 형 배열을 선언해라' 문장을 Arrays 라고 하나의 이름으로 바꾼 것이라 보면 됩니다. 즉,

```
int arr[10];
Arrays arr;
```

은 정확히 동일한 문장입니다.

volatile 키워드

volatile 은 아주 아주 특수한 상황이 아니고서는 사용하지 않는 키워드 입니다. 사실 저도 그렇게 자주 사용하지 않고요. volatile 키워드를 사용하는 경우는 대부분 외부 하드웨어와 통신할 때 사용하게 됩니다. 이 말이 무슨 말이냐면, 아래 예제를 보시면 알게 될 것입니다.

만일 여러분이 특정한 외부 센서와 소통하는 프로그램을 만든다고 합시다. 이 센서는 RAM 의 특정 영역을 이용하는데, 만일 센서에 값이 감지되지 않으면 그 곳의 값이 0 이 되어 무언가가 감지되면 그 부분의 값을 1 로 한다고 합시다. 그렇다면 여러분은 십중팔구 아래와 같은 코드를 작성할 것입니다.

```c
#include <stdio.h>
typedef struct SENSOR {
  /* 감지 안되면 0, 감지되면 1 이다.*/
  int sensor_flag;
  int data;
} SENSOR;

int main() {
  SENSOR *sensor;
  /* 값이 감지되지 않는 동안 계속 무한 루프를 돈다*/
  while (!(sensor->sensor_flag)) {
  }
  printf("Data : %d \n", sensor->data);
}
```

참고로 typedef 를 위와 같이 써줌으로써 typedef struct SENSOR SENSOR 한 효과를 낼 수 있습니다.

위 코드는 가상의 코드 이므로 컴파일 해보지 않겠습니다만, 일단 여러분은 위 코드에서 별 이상을 느끼지는 못할 것입니다. 하지만 똑똑한 컴파일러는 '너무 과하게 똑똑해서' 우리가 사용한 while 문을 최적화 해버립니다. 보통의 상황에서 sensor->sensor_flag 의 값이 바뀌는 경우는 없기 때문에 굳이 while 문을 매번 돌릴 때 마다 값을 비교할 필요가 없게 되는 것이지요. 그냥 컴파일러는 값을 딱 한 번만 읽고 0 이 아니라면 그냥 가고, 0 이라면 while 문을 무한히 돌리는 것으로 생각해버립니다. 결과적으로 위 코드를 컴파일러는 다음과 같은 코드로 바꿔버립니다.

```c
#include <stdio.h>
typedef struct SENSOR {
  /* 감지 안되면 0, 감지되면 1 이다.*/
  int sensor_flag;
  int data;
} SENSOR;
int main() {
  SENSOR *sensor;
  if (!(sensor->sensor_flag)) {
    while (1) {
    }
  }
  printf("Data : %d \n", sensor->data);
}
```

이는 우리가 결코 원하던 결과가 아닙니다. 만일 센서에 값이 감지되었다고 해도 while 문을 절대로 탈출할 수 없게 되어 무한 루프에 빠지게 되는 것이지요. 우리는 컴파일러가 이런 최적화 작업을 수행하는 것을 원하지 않습니다. 이를 컴파일러에게 알려주기 위해서는 두 가지 방법이 있습니다.

첫번째로는 컴파일러의 최적화 옵션을 빼버리는 것입니다. gcc 에서는 단순히 최적화 옵션을 안 주면 됩니다. Visual Studio 에서는 살짝 복잡한데, 프로젝트 속성의 **C/C++ -> 최적화**

에서 사용 안함을 선택하시면 됩니다. 그런데, 최적화를 하지 않기에는 너무나 그 손실이 큽니다. 최적화 옵션을 끄는 순간 다른 모든 코드들도 최적화를 하지 않겠다는 의미가 되거든요. 이를 위해 volatile 키워드가 생겨났습니다.

```
#include <stdio.h>
typedef struct SENSOR {
  /* 감지 안되면 0, 감지되면 1 이다.*/
  int sensor_flag;
  int data;
} SENSOR;

int main() {
  volatile SENSOR *sensor;
  /* 값이 감지되지 않는 동안 계속 무한 루프를 돈다*/
  while (!(sensor->sensor_flag)) {
  }
  printf("Data : %d \n", sensor->data);
}
```

이렇게 해준 순간 컴파일러는 sensor 에 대해 최적화를 수행하지 않게 됩니다. volatile 의 의미는 '변덕스러운' 이라는 의미를 가지고 있는데, sensor 에 volatile 키워드를 붙여준 순간 sensor->sensor_flag 의 값이 '변덕스럽게 변할 수 있기 때문' 에 이에 대한 최적화 작업들을 수행하지 말라 라는 의미가 됩니다. 따라서 컴파일러는 위 소스를 의미 그대로 컴파일 하게 되어 우리가 원하던 결과를 얻을 수 있게 됩니다.

#pragma 키워드

#pragma 는 **컴파일러에게 말하는 전처리기 명령** 이라고 보시면 됩니다. 즉, #include 나 #define 처럼 전처리기에 의해 컴파일 이전에 처리되지만, 그 명령은 컴파일러에게 전달되기 때문이죠. 사실 pragma 는 C 언어의 기본 키워드라고 하기 보다는, 컴파일러에 종속적인 키워드라고 하는 것이 맞습니다. pragma 를 사용하는 문법은 컴파일러 마다 다르고 딱히 통일 된 것이 없기 때문입니다. 이 강좌에서는 pragma 를 사용하는 몇 가지 예제들을 보고 어떤 경우에 편리하게 pragma 를 사용할 수 있는지 살펴봅시다.

#pragma pack

```
#include <stdio.h>
struct Weird {
  char arr[2];
  int i;
};
int main() {
```

```
  struct Weird a;
  printf("size of a : %d \n", sizeof(a));
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
size of a : 8
```

와 같이 나옵니다.

상당히 이상하지요. 분명히 Weird 구조체 내의 원소들의 총 바이트 수를 계산해보면 arr 은 char 형 변수 2 개로 2 바이트 이고, i 는 int 형 변수 1 개로 4 바이트 이므로 6 이 나와야 정상이지요. 그런데 도대체 왜 컴퓨터는 이를 8 로 출력했을까요?

왜냐하면 실제로 메모리 상에서 위 구조체의 크기를 8 바이트로 **컴파일러**가 지정하였기 때문입니다. 현재 우리가 사용하는 컴퓨터에서는 언제나 4 바이트 단위로 모든 것을 처리하는 것이 빠릅니다. 따라서 언제나 컴퓨터에서 데이터를 보관할 때 에는 4의 배수로 데이터를 보관하는 것이 처리시 용이하게 됩니다. 이렇게 데이터가 4 의 배수 경계에 놓인 것을 더블 워드 경계에 놓여 있다 라고 합니다.

이러한 이유 때문에 위 Weird 구조체 역시 4 의 배수를 맞추기 위해 크기를 8 바이트로 '필요없는 2 바이트를 추가하면서 까지' 맞춘 것입니다. 이 문제가 중요하게 여겨지는 부분은 역시 하드웨어 간의 통신 때문에 그렇습니다.

예를 들어서 SCSI 인터페이스는 PC 에서 하드 디스크와 같은 주변 기기에 연결하기 위한 통신 방식으로 SCSI 장치들에게 읽기 명령을 내리기 위해서는 6 바이트의 명령어를 전송하면 됩니다. 이 6 바이트의 명령어의 구조는 꽤 복잡해서 흔히 구조체로 많이 이용하는데, 만일 위와 같이 그냥 사용했다가는 구조체의 크기가 8 바이트로 설정되어서 무슨 문제가 생길 지 알 수 없습니다.

이렇게 컴파일러로 하여금 구조체를 더블 워드 경계에 놓지 말라고 하고 싶을 때 pragma 키워드를 이용하면 됩니다.

```
#include <stdio.h>
/* 전처리기 명령에는 ; 를 붙이지 않는다! */
#pragma pack(1)
struct Weird {
  char arr[2];
  int i;
};
int main() {
  struct Weird a;
  printf("size of a : %d \n", sizeof(a));
```

```
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
size of a : 6
```

와 같이 나옵니다.

이번에는 6 으로 잘 나옵니다. 위 명령은 마이크로소프트 계열의 컴파일러들에만 유효한 문장인데, 구조체를 '1 바이트 단위로 정렬하라는 뜻' 입니다. 즉, 구조체의 크기가 1 의 배수가 되게 하라는 것이지요. 1 외에도 2,4,8,16 등이 올 수 있습니다. 만일 기본값, 즉 더블 워드 경계로 정렬하기 위해서는 `#pragma pack(4)` 로 하시면 됩니다.

#pragma once

아까의 `Weird` 구조체 예제에서 `Werid` 부분만 다른 헤더파일로 빼놓아 봅시다. 이 헤더파일의 이름은 `werid.h` 입니다.

```
/* weird.h */
struct Weird {
  char arr[2];
  int i;
};
```

```
/* test.c*/
#include <stdio.h>
#include "weird.h"
int main() {
  struct Weird a;
  a.i = 3;
  printf("Weird 구조체의 a.i : %d \n", a.i);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
Weird 구조체의 a.i : 3
```

와 같이 나옵니다.

상당히 단순한 예제이지요. `test.c` 에서 `weird.h` 를 포함했으므로 `weird.h` 의 내용이 `test.c` 로 그대로 복사된 셈입니다. (즉, #include "weird.h" 부분이 `weird.h` 의 내용으로 바뀌었다고 보셔도 무방합니다) 따라서 `struct Weird` 를 사용할 수 있게 되므로 위와 같은 결과가 발생합니다. 그런데 만일 실수로 `weird.h` 를 두 번 포함했다고 합시다. 그렇다면 어떻게 될까요?

```c
#include <stdio.h>
#include "weird.h"
#include "weird.h"
int main() {
  struct Weird a;
  a.i = 3;
  printf("Weird 구조체의 a.i : %d \n", a.i);
  return 0;
}
```

컴파일 하였다면

컴파일 오류

```
error C2011: 'Weird' : 'struct' 형식 재정의
 'Weird' 선언을 참조하십시오.
```

위와 같이 오류를 만나게 됩니다. 왜냐하면 각각 `#include "weird.h"` 부분이 `weird.h` 의 내용으로 바뀌어서 결과적으로는

```c
#include <stdio.h>
struct Weird {
  char arr[2];
  int i;
};
struct Weird {
  char arr[2];
  int i;
};

int main() {
  struct Weird a;
  a.i = 3;
  printf("Weird 구조체의 a.i : %d \n", a.i);
  return 0;
}
```

를 한 것과 마찬가지가 되어서 `struct Weird` 를 두 번 정의하였다고 오류가 나게 됩니다. 이를 막으려면 어떻게 해야 할까요?일단 C 의 기본 전처리기 명령을 이용하여 하는 방법이 있습니다.

```c
/* 수정된 weird.h*/
#ifndef WEIRD_H
#define WEIRD_H
struct Weird {
  char arr[2];
  int i;
};
#endif
```

```c
/* 이상한 test.c*/
#include <stdio.h>
#include "weird.h"
int main() {
  struct Weird a;
  a.i = 3;
  printf("Weird 구조체의 a.i : %d \n", a.i);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Weird 구조체의 a.i : 3
```

와 같이 잘 실행됩니다. 일단 왜 오류가 나지 않는지 살펴 봅시다. 우리가 전처리기라고 한다면 맨 처음에 첫번째 `#include "weird.h"` 를 만났을 때 `WEIRD_H` 가 정의되어 있지 않으므로 `#ifndef` 가 참이 되어 아래 `#define WEIRD_H` 가 수행되어 `WEIRD_H` 라는 것이 정의됩니다. (값은 모르지만 아무튼, 이러한 이름이 정의되었다고 합시다)

또한 헤더파일의 내용도 `test.c` 로 그대로 복사되죠. 그 후에 실수로 `weird.h` 를 다시 한 번 `include` 하였을 때 에는 이미 `WEIRD_H` 가 정의되어 있는 상태이므로 `#ifndef WEIRD_H` 가 거짓이 되어 `#endif` 로 넘어가버려 `test.c` 에 그 내용이 복사가 안됩니다.

이렇게 하면 헤더파일의 내용이 중복으로 포함되는 것을 막을 수 있습니다. (이는 이미 수많은 헤더파일에서 사용되고 있는 방법입니다) 하지만 `#pragma` 를 이용하면 훨씬 단순하게 할 수 있는데,

```c
/* #pragma 의 위엄 - weird.h*/
#pragma once
struct Weird {
  char arr[2];
  int i;
};
```

```c
/* test.c*/
#include <stdio.h>
#include "weird.h"
int main() {
  struct Weird a;
  a.i = 3;
  printf("Weird 구조체의 a.i : %d \n", a.i);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Weird 구조체의 a.i : 3
```

와 같이 잘 나옵니다. 이 명령은 컴파일러로 하여금 이 파일이 오직 딱 한 번만 include 될 수 있다는 것을 말해주는데, 이는 위에서 #ifndef 를 이용하여 복잡하게 하였던 작업들을 단순하게 한 문장으로 끝낼 수 있게 됩니다.

또한 #pragma once 의 장점으로 #ifndef 를 이용하는 것 보다 컴파일 시간을 절약할 수 있다는 점인데, #ifndef 를 이용하게 되면 include 하였을 때 전처리기가 직접 헤더파일을 열어 보아서 과연 WEIRD_H 가 정의되었나 정의되지 않았나 확인해 보아야 하는데, pragma once 를 이용하면 한 번 include 되었다면 헤더파일을 다시 열어보지도 않기 때문에 컴파일 시간이 절약되는 효과가 나게 됩니다.

다만 앞에서도 말했듯이 #pragma 관련 키워드들이 컴파일러 종속적이여서 어떤 컴파일러에서는 #pragma once 가 지원이 되지 않을 수 도 있습니다. 따라서 무슨 컴파일러를 사용하는지 보고 #pragma once 를 지원한다면 되도록 이것을 사용하는 것이 도움이 됩니다.

실제로 아래 코드는 stdio.h 의 헤더파일을 열어본 것입니다.

```c
/***
 *stdio.h - definitions/declarations for standard I/O routines
 *
 *       Copyright (c) Microsoft Corporation. All rights reserved.
 *
 *Purpose:
 *       This file defines the structures, values, macros, and functions
 *       used by the level 2 I/O ("standard I/O") routines.
 *       [ANSI/System V]
 *
 *       [Public]
 *
 ****/
```

```
#if _MSC_VER > 1000
#pragma once
#endif

#ifndef _INC_STDIO
#define _INC_STDIO

/* 내용 (생략) */

#endif /* _INC_STDIO */
```

위 헤더파일에서 사용하는 컴파일러마다 어떠한 키워드를 사용할 수 있게 하였는지 알 수 있는데,

```
#if _MSC_VER > 1000
#pragma once
#endif
```

를 보면 _MSC_VER 이 1000 보다 크면 #pragma once 키워드를 사용하라고 되어있습니다. _MSC_VER 은 마이크로소프트 사의 전처리기에 의해 기본적으로 정의되어 있는 상수로 컴파일러의 버전을 나타내는데, Visual C++ 의 경우 _MSC_VER 값이 1000 부터 시작 하여 현재 2008 버전은 1500 의 값을 가지고 있습니다. 즉, 현재 버전의 컴파일러의 경우 _MSV_VER > 1000 이 참이 되므로 #pragma once 키워드를 이용하게 됩니다. 구 버전의 컴파일러는 그 아래

```
#ifndef _INC_STDIO
#define _INC_STDIO

...

#endif /* _INC_STDIO */
```

과 같이 C 표준 방식의 형태를 사용하도록 되어 있는 것을 볼 수 있습니다.

자. 이제 C 언어의 가장 뒷부분인 typedef, volatile, #pragma 와 같은 키워들에 대해서도 모두 알아 보았습니다. 이제 더이상 강의할 내용들도 별로 없는 것 같아서 슬프네요 ㅠㅠ. 아무튼 제 강의를 읽는 모든 분들, 즐거운 성탄절을 보내시기 바랍니다 :)

생각해 볼 문제

문제 1

MSDN 에 들어가서 #pragma 와 연관된 키워드들을 잘 살펴보시기 바랍니다.http://msdn.microsoft.com/en-us/library/d9x1s805%28v=VS.71%29.aspx

21

파일 입출력

안녕하세요~ 여러분 드디어 38 번째 강좌 입니다! 제 목표로는 40 번째 강좌를 끝으로 마칠 예정인데 조금 더 길어 질지도 모르겠군요 :) 여러분이 여태까지 프로그램들을 만들면서 '데이터를 어떻게 하면 프로그램이 종료되어도 보관할 수 있을까?' 라는 생각을 많이 하셨을 것입니다. 사실 그 방법은 단순합니다. 특정한 데이터가 있으면 이를 하드디스크에 기록하면 해결되는 일이지요.

여태까지 만든 모든 프로그램에서 변수는 하드디스크가 아니라 언제나 RAM 에 상주하는 데이터 였습니다. 즉, 프로그램이 종료되어도 그렇지만 컴퓨터가 꺼지게 되면 데이터가 날아가게 되는 **휘발성 메모리** 이지요. (여러분이 배운 내용이 이렇지 않기를 바랍니다) 하지만 여러분의 컴퓨터에 깔려있는 대부분의 프로그램이나 문서들은 껐다 켜도 사라지지 않습니다. 왜냐하면 그 내용들이 **비휘발성 저장매체** 인 하드 디스크에 저장되어 있기 때문입니다.

그렇다고 해서 하드 디스크에 아무렇게나 데이터를 보관할 수 있는 것은 아닙니다. 하드디스크에 데이터를 보관할 때 에는 **파일**의 단위로 데이터를 보관하게 됩니다. 따라서 이번 강좌에서는 어떻게 하면 파일을 만들고, 파일에 데이터를 저장하고, 파일을 읽어들일 수 있을지 알아보도록 하겠습니다.

파일에 출력하기

```c
/* a.txt 에 내용을 기록한다. */
#include <stdio.h>

int main() {
  FILE *fp;
  fp = fopen("a.txt", "w");

  if (fp == NULL) {
    printf("Write Error!!\n");
    return 0;
  }

  fputs("Hello World!!! \n", fp);
```

```
  fclose(fp);
  return 0;
}
```

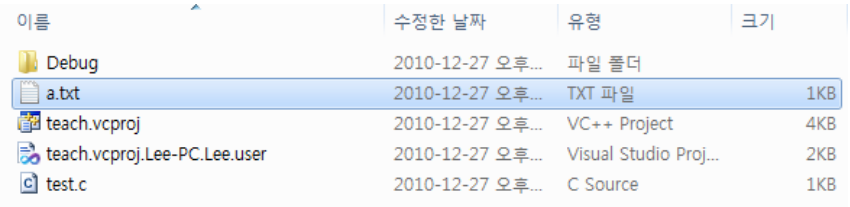
성공적으로 컴파일 하였다면

실행 결과

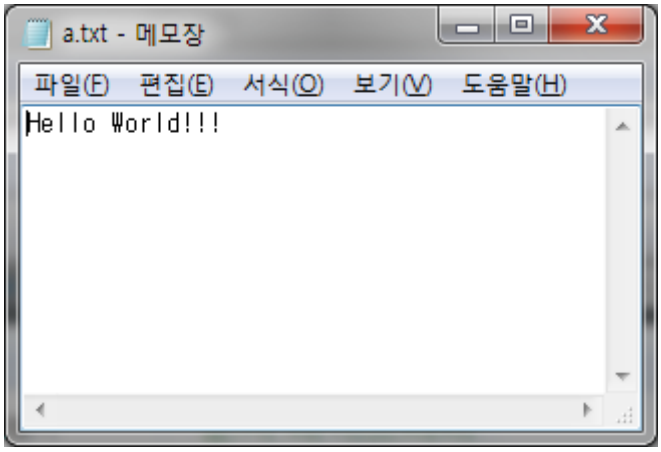
와 같이 아무것도 나오지 않습니다. 왜냐하면 화면에 출력하는 문장이 아무것도 없거든요. 대신, 소스 파일이 위치한 곳으로 들어가봅시다. 저의 경우 다음과 같은 경로에 소스파일이 위치해있습니다.

C:\Users\Lee\Documents\Visual Studio 2008\Projects\teach\teach

찾으셨다면 아래 그림처럼 예쁘게 `a.txt` 라는 파일이 생성된 것을 볼 수 있습니다.



그렇다면 기대되는 마음으로 이 파일을 열어보겠습니다.



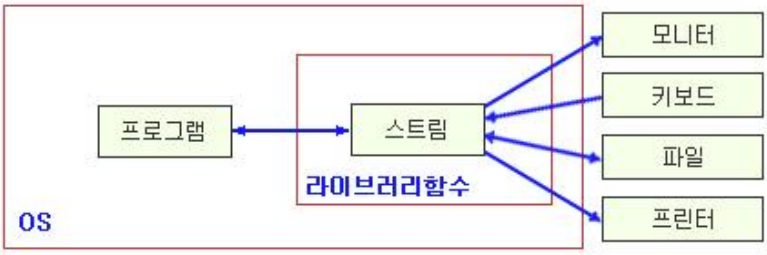
와우! 우리가 원하던 문자열 "Hello World!!!" 가 제대로 들어가 있는 것을 보실 수 있습니다. 이제, 다시 소스 코드를 살펴보도록 하죠.

```
FILE *fp;
fp = fopen("a.txt", "w");
```

사실 우리가 하드디스크에 저장되어 있는 파일들을 자유롭게 이용할 수 있다고는 하나 이를 쓰는 과정은 매우매우 복잡할 것입니다. 왜냐하면 파일을 새로 만든다고 쳐도, 하드디스크 어떤 부분에 파일을 새로 만들어야 할지, 얼마나 크게 파일을 만들 수 있는지 등의 모든 것들을 고려해야 합니다. 자그마한 파일 하나를 만드는데 이런 짓들을 하기엔 너무 지나친 일이지요. 그래서 다행스럽게도 이와 같은 복잡한 일들은 컴퓨터 운영체제에서 알아서 해줍니다.

`fopen` 함수는 바로 위에서 말한' 운영체제가 알아서 해주는 부분' 을 처리합니다. `fopen` 함수는 우리가 지정한 파일(a.txt) 과 소통할 수 있도록 스트림을 만들어 줍니다. 어, 그렇다면 스트림이 무엇일까요?

스트림



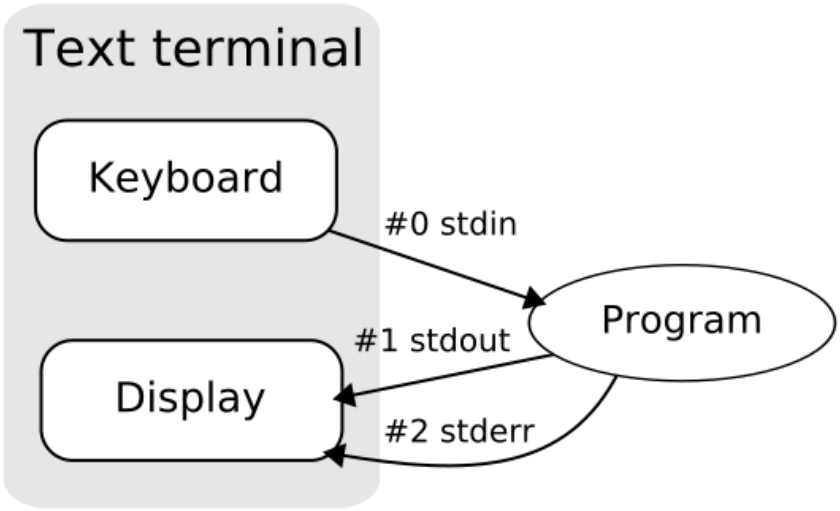
우리가 `printf` 함수를 이용할 때 어떠한 작업이 컴퓨터에서 내부적으로 처리되는지 생각해봅시다. 먼저, 출력할 문자열을 구성해야 겠죠. 그리고 이를 모니터에 전달해서 출력하라는 명령을 내리게 해야 합니다. 과연 이것이 쉬운 일일까요? 모니터에 명령을 내리기 위해서는 모니터를 만든 회사마다 그 방식이 다를 것이고, 어떠한 명령을 내려햐 하는지도 다를 것입니다. 하지만 우리는 이를 `printf` 라는 함수 하나로 이 모든 것을 할 수 있었습니다.

그 이유는 바로 **스트림** 에 있습니다. 스트림은 이 두 개의 완전히 다른 장치들을 이어주는 **파이프** 라고 보시면 됩니다. 이러한 스트림은 우리가 직접 구현해야 되는 것이 아니라 운영체제가 스스로 처리해 주는 것이지요. 만일 우리가 모니터와 잇는 스트림을 이용한다면 운영체제는 모니터에 맞는 명령을 내릴 것이고, 키보드와 잇는 스트림을 이용한다면 운영체제가 키보드에 맞는 명령을 알아서 내릴 것입니다. 우리 프로그래머 입장에서는 걱정을 전혀 할 필요가 없겠지요.[1)]

따라서 만일 우리가 모니터에 A 를 출력하고 싶다면 단순히 스트림에 A 를 넣으면 됩니다. 왜냐하면 이렇게 스트림으로 전달된 문자 A 는 운영체제에 의해 알아서 모니터에 명령을 내려서 A 를 출력하게 되지요. 마찬가지로 키보드에서 문자를 받고 싶다면 스트림을 타고 무슨 문자가 오는지에만 관심을

1) 수준 높은 분들을 위한 설명 : 스트림을 정확히 표현하면 '추상화된 장치(abstract devices)' 라고 말할 수 있습니다. 왜냐하면 여러가지 주변 장치들(모니터, 키보드, 하드 디스크) 을 추상화 시켜서 사용자가 마치 동일한 장치에 접근하는 것 처럼 사용할 수 있게 만들었기 때문이죠. 어떠한 모습으로 장치들을 추상화 시켰냐면 스트림은 마치 책장과 같이 만들었습니다. 책장에 책을 끼우거나 빼는 것 처럼 데이터를 순차적으로 쭉 나열해서 데이터의 끝 까지 차례대로 읽어들일 수 있도록 만들었습니다.

가지면 됩니다. 왜냐하면 우리가 키보드에 무언가를 입력했다면 운영체제에서 알아서 잘 해석을 한 다음 우리가 이해할 수 있는 데이터로 만들어서 스트림에 전달하기 때문이죠.



http://en.wikipedia.org/wiki/File:Stdstreams-notitle.svg

그런데 사실 생각해보면 우리는 위에서 말한 두 예인 모니터와 키보드에 대한 스트림을 한번도 만든 적이 없습니다. 파일을 이용할 때 에는 파일에 대한 스트림을 `fopen` 으로 만든다고 했는데 말이죠. 사실 모니터와 키보드에 대한 스트림은 **표준 스트림(standard stream)** 이라 해서 프로그램이 실행될 때 자동으로 생성됩니다.

위 그림에도 달 나와있듯이 모니터에 대한 스트림은 `stdout` 이고, 키보드에 대한 스트림은 `stdin` 입니다. (그 외에 `stderr` 이라는 표준 오류 스트림이란 것이 있는데 `stdout` 하고 거의 동일하다고 보시면 됩니다. 단지, 오류 메세지를 출력하는 스트림 입니다)

이제 다시 맨 처음의 예제로 돌아가보도록 합시다.

```
FILE *fp;
fp = fopen("a.txt", "w");
```

이렇게 해서 스트림을 만들었으면 `fopen` 함수는 만든 스트림을 가리키는 포인터를 리턴합니다. 스트림에 관한 정보는 `FILE` 구조체에 들어가 있습니다. (FILE 구조체에 대한 자세한 내용을 알고 싶다면 여기로) 이제, 우리는 `fp` 를 가지고 파일을 사용할 수 있게 되는 것입니다. 그런데, 우리는 `fopen` 에서 두 번째 인자로 "w" 를 전달했는데, 이 말의 의미는 파일에 오직 '쓰기' 만이 가능하게 하겠다 라는 의미 입니다. 다시 말해 스트림인데도 출력 스트림만 만들어 놓은 것이지요. (파일에 쓰는 것은 프로그램의 관점에서 보았을 때 출력이므로 출력 스트림, 파일에서 읽는 것은 프로그램의 관점에서 보았을 때 입력 받는 것이므로 입력 스트림 입니다) 쉽게 말하면 일방 통행 도로를 만들어 놓은 것과 같습니다.

이렇게 출력만 하게 했다면 당연히 파일에 **쓰기** 만 할 수 있습니다. 파일에서 데이터를 읽는 작업은 불가능 하게 됩니다. 일단 읽는 것은 나중에 생각하기로 하고 어떻게 파일에 쓰기를 하는지 알아보도

록 합시다. fopen 에서 "w" 로 전달했을 때 특징이, 첫번째 인자로 전달된 이름의 파일이 존재하지 않는다면 아무 내용이 없는 파일을 새로 만들거나, 동일한 이름의 파일이 존재 한다면 그 내용을 다 지워버리게 됩니다. 참고로, "a.txt" 로 그냥 파일의 이름을 전달한다면 오직 '소스 파일과 동일한 경로에 들어있는 파일들' 을 찾게 됩니다. 만일 다른 폴더에 있는 a.txt 를 찾고 싶다면 그 경로를 넣어주면 됩니다.

예를 들어 C 드라이브의 BBB 라는 폴더의 a.txt 를 원한다면 다음과 같이 하면 됩니다.

```
fp = fopen("C:\\BBB\\a.txt", "w");
```

이 때 \\ 를 쓰는 이유는 \ 하나만 쓰면 escape character 라고 해서 이상한 문자가 되므로 \\ 를 두개 붙여 써서 \ 로 나타내야 합니다.

아무튼, 우리의 a.txt 의 경우 원래 존재 하지 않았을 것이므로 fopen 에서 a.txt 를 "w" 로 여는 순간 새로운 파일이 만들어집니다.

```
if (fp == NULL) {
  printf("Write Error!!\n");
  return 0;
}
```

이 다음은 아주 중요한 부분인데, 파일이 어떠한 이유에서라든지 열지 못한 경우 fopen 함수는 NULL 을 리턴합니다. fopen 이 실패하는 경우는 그리 많지 않으므로 이 부분을 생략하는 경우가 가끔 있는데, 만일 fopen 이 실패하게 되었을 경우 이렇게 검사하지 않는다면 소스 뒷부분에서 어떠한 문제가 발생할지 모르므로 이렇게 항상 검사하는 것이 중요합니다.

```
fputs("Hello World!!! \n", fp);
```

이제 fputs 라는 훌륭한 함수로 파일에 기록할 수 있습니다. 첫번째 인자로 파일에 기록할 문자열을 전달하고 두번째 인자로 어떠한 스트림을 택할지 그 포인터를 써주면 됩니다. 우리는 우리가 위에서 열은 파일 스트림을 택할 것이므로 fp 를 써주면 됩니다. 재미있는 사실은 표준 스트림들은 이미 이름이 정해져 있는데 앞서 말했듯이 stdout 은 컴퓨터의 모니터에 해당하는 표준 출력 스트림이라 했습니다. 즉, 두 번째 인자로 stdout 을 전달하면 우리 콘솔 화면에 그 문자열이 뜨게 되겠지요.

```
fputs("Hello World!!! \n", stdout);
```

을 해보면

실행 결과

```
Hello World!!!
```

와 같이 실제로 잘 나오는 것을 알 수 있습니다. 아무튼

```
fputs("Hello World!!! \n", fp);
```

를 통해 파일에 "Hello World!!! \n" 을 기록하게 됩니다. 이제 마지막으로

```
fclose(fp);
```

를 통해 연결되었던 스트림을 닫아 주어야만 합니다. 만일 이렇게 fclose 로 닫지 않는다면 스트림이 계속 살아 있게 되어서 이 파일은 계속 쓰기 상태로 남아 있게 됩니다. 이는 프로그램이 종료되기 전까지 이 상태로 계속 남아 있기 때문에, 마치 동적 메모리 할당에서 free 로 메모리를 반환해 주어야 하는 것처럼 스트림도 닫아 주어야 합니다.

재미있는 사실은 fclose 로 표준 스트림들도 닫아버릴 수 있는데 예를 들어

```
/* stdout 을 닫아버린다 */
#include <stdio.h>
int main() {
  fclose(stdout);
  printf("aaa");
  return 0;
}
```

으로 표준 출력 스트림을 닫아버리면

실행 결과

```

```

와 같이 printf 를 해도 아무것도 나오지 않는 재미있는 일이 발생합니다.

파일에서 입력 받기

```
/* fgets 로 a.txt 에서 내용을 입력 받는다. */
```

```c
#include <stdio.h>
int main() {
  FILE *fp = fopen("a.txt", "r");
  char buf[20];  // 내용을 입력받을 곳
  if (fp == NULL) {
    printf("READ ERROR !! \n");
    return 0;
  }
  fgets(buf, 20, fp);
  printf("입력받는 내용 : %s \n", buf);
  fclose(fp);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
Hello World!!
```

한 번 소스코드를 살펴봅시다.

```c
FILE *fp = fopen("a.txt", "r");
```

이번에는 "w" 가 아니라 "r" 형으로 열었습니다. 이번에는 **읽기** 형식으로 파일을 열게됩니다.

```c
if (fp == NULL) {
  printf("READ ERROR !! \n");
  return 0;
}
```

이전 예제와 마찬가지로 fp 가 NULL 인지 아닌지 확인하는데, 특히 읽기 형식으로 파일을 열 때에는 더욱 주의해야 할 부분입니다. 왜냐하면 쓰기 형식으로 파일을 열었을 때 에는 파일이 존재하지 않는다면 새로 만들었지만 읽기 형식으로 열 때 에는 읽어들일 파일이 없다면 NULL 을 리턴하고 스트림을 만들지 않기 때문이지요.

```c
fgets(buf, 20, fp);
```

이제 fgets 함수를 통해 파일로 부터 문자열을 입력 받습니다. 첫번째 인자로 어디에 입력받을지, 두번째 인자로 입력받을 바이트 수, 세번째 인자로 어떤 스트림을 통해 입력받을지 명시해 주면 됩니다.

우리의 경우 buf 라는 공간에 20 바이트를 입력받을 것입니다. fgets 의 좋은 점이 입력받는 양을 제한할 수 있다는 점인데 기존의 scanf 와 의 경우 문자열을 입력 받을 때 제한을 두지 않아 할당된 메모리 크기를 넘어버리는 오버플로우 (예를 들어 char str[20]; 에 100 글자를 입력 받는다던지) 가 되는 경우가 있었지만 fgets 는 이를 방지할 수 있으므로 상당히 안정적이라고 볼 수 있습니다.

```
printf("입력받는 내용 : %s \n", buf);
```

이렇게 입력 받은 printf 로 출력하면 됩니다.

```
/* 한 글자씩 입력받기*/
#include <stdio.h>

int main() {
  FILE *fp = fopen("a.txt", "r");
  char c;

  while ((c = fgetc(fp)) != EOF) {
    printf("%c", c);
  }

  fclose(fp);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
Hello World!!
```

와 같이 나옵니다.

```
while ((c = fgetc(fp)) != EOF) {
  printf("%c", c);
}
```

주목할 부분은 위 부분 입니다. fgetc 는 fp 에서 문자 하나를 얻어옵니다. 즉, 한 문자씩 읽어들이는 것이지요. 이 때 문자열 맨 마지막이 NULL 문자로 종료를 나타내는 것 처럼,파일의 맨 마지막에는 EOF 라고 End Of File 을 나타내는 값인 -1 이 들어가 있습니다. 실제로 EOF 의 원형을 찾아보아도

```
#define EOF (-1)
```

로 -1 로 선언되어 있습니다. 따라서 우리는 c 가 EOF 인지 아닌지 비교함을 통해 파일의 끝까지 입력을 받았는지 안받았는지 알 수 있습니다. 이와 같은 방식을 통해 아래 예제 처럼 파일의 크기를 알아내는 프로그램도 만들 수 있습니다.

```c
#include <stdio.h>

int main() {
  FILE *fp = fopen("a.txt", "r");
  int size = 0;

  while (fgetc(fp) != EOF) {
    size++;
  }

  printf("이 파일의 크기는 : %d bytes \n", size);
  fclose(fp);
  return 0;
}
```

성공적으로 컴파일 했다면

실행 결과

```
이 파일의 크기는 : 14 bytes
```

와 같이 잘 나옵니다.

원리는 이전의 예제와 동일합니다. EOF 가 나오기 전 까지 계속 size 를 증가시켜서 파일의 크기를 알아내는 것이지요.

파일 위치 지정자

여태까지 파일에서 입력을 받을 때 언제나 파일의 시작 부분에서 끝 부분으로 입력을 쭉 받아 나갔습니다. 즉, 이전에 입력 받았던 데이터는 다시 입력 받지 않았다는 것이지요. 이것이 가능하게 된 이유는 **파일 위치 지정자** 때문 입니다. 영어로 **Position Indicator** 라고 합니다.

만일 a.txt 에 abcdefg 가 들어있고 우리가 fgetc 로 입력을 받는다고 해봅시다. 파일을 맨 처음 열었을 때 에는 파일 위치 지정자는 파일의 맨 첫부분을 가리키고 있습니다. 따라서 a 를 가리키고 있다고 보아도 무방합니다. 이제, 우리가 fgetc 로 입력을 받는다면 파일 위치지정자는

한 칸 넘어가서 다음에 입력 받을 것을 가리키고 있게 되지요. 따라서 fgetc 를 한 번 더하면 a 를 다시 입력 받는 것이 아니라 그 다음인 b 를 입력 받게 됩니다. 그리고 또 파일 위치지정자는 또 한 칸 이동해서 그 다음인 c 를 가리키고 있겠지요.

그런데 만일 여러분이 abcd 까지 파일에서 입력 받았는데 다시 처음 부터 입력받고 싶다면 어떻게 할까요? 일단 두 가지 방법이 있는데 하나는 fopen 으로 파일을 다른 스트림으로 또 여는 것이고, 또다른 방법은 파일 위치지정자를 맨 앞으로 옮기면 되겠지요. 여기서는 후자를 택하도록 합시다.

```
#include <stdio.h>
int main() {
  /* 현재 fp 에 abcdef 가 들어있는 상태*/
  FILE *fp = fopen("a.txt", "r");
  fgetc(fp);
  fgetc(fp);
  fgetc(fp);
  fgetc(fp);
  /* d 까지 입력받았으니 파일 위치지정자는 이제 e 를 가리키고 있다 */
  fseek(fp, 0, SEEK_SET);
  printf("다시 파일 처음에서 입력 받는다면 : %c \n", fgetc(fp));
  fclose(fp);
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
다시 파일 처음에서 입력 받는다면 : a
```

와 같이 a 가 다시 잘 나오는 것을 보실 수 있습니다.

```
fgetc(fp);
fgetc(fp);
fgetc(fp);
fgetc(fp);
```

일단 a.txt 에 원래 abcdef 가 들어있었다고 합시다. 그렇다면 위 문장을 통해 차례대로 a,b,c,d 를 입력받고 (물론 저장은 하지 않지만) 이제 파일 위치지정자는 e 를 가리키게 됩니다. 그런데,

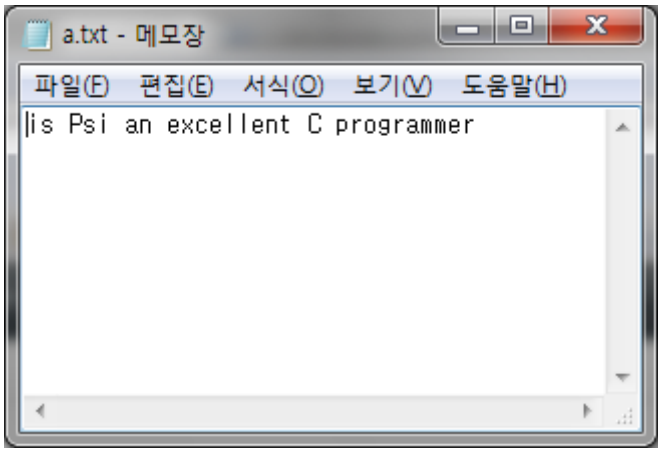
```
fseek(fp, 0, SEEK_SET);
```

를 통해 파일 위치지정자를 맨 처음으로 돌려버릴 수 있었습니다. fseek 함수는 fp 를 세번째 인자로 부터 두번째 인자 만큼 떨어진 곳으로 파일 위치지정자를 돌리는데, 위 경우 SEEK_SET 으로 부터 0 번째 떨어진 곳, 즉 SEEK_SET 으로 돌린다고 볼 수 있습니다.

이 때 SEEK_SET 은 파일의 맨 처음을 일컫는 매크로 상수 입니다. 따라서 위 함수를 통해 fp 의 파일 위치지정자를 맨 처음으로 돌려서 다시 fgetc 를 하였을 때 a 를 입력받게 됩니다. 참고로, SEEK_SET 외에도, 현재의 위치를 표시하는 SEEK_CUR 과 파일의 맨 마지막을 표시하는 SEEK_END 상수들이 있습니다.

```c
/* 출력 스트림도 마찬가지*/
#include <stdio.h>
int main() {
  FILE *fp = fopen("a.txt", "w");
  fputs("Psi is an excellent C programmer", fp);
  fseek(fp, 0, SEEK_SET);
  fputs("is Psi", fp);
  fclose(fp);
  return 0;
}
```

성공적으로 컴파일 하였을 때, a.txt 의 모습을 보면



로 나타납니다. 사실 이번 예제도 상당히 쉬운데, 먼저 fputs 로

```c
fputs("Psi is an excellent C programmer", fp);
```

Psi is an excellent C programmer 을 넣었고, 이 때 파일을 열어보았더라면 이와 같은 문장이 들어 있었을 것입니다. 그런데,

```c
fseek(fp, 0, SEEK_SET);
```

로 파일 위치지정자를 맨 처음으로 돌려서 다시 fputs 를 했을 때, 파일 앞에 내용이 끼워져 들어가는 것이 아니라 이전의 내용에 덮어쓰기 하면서 기록이 되므로 맨 처음 *Psi is* 를 *is Psi* 로 내용을

바꿔버립니다. 따라서 결국에는 *is Psi an excellent C programmer* 라는 문장이 파일에 남아 있게 됩니다.

이번 강좌에서는 이렇게 대략적으로 파일 입출력을 어떻게 하는 것인지, 그리고 파일 위치지정자가 무엇인지 소개했습니다. 사실 파일 입출력의 백미는 다음 강좌에서 부터 시작이라 보시면 됩니다 :)

생각해보기

문제 1

사용자로 부터 경로를 입력 받아서 그 곳에 파일을 생성하고 a 를 입력해놓는 프로그램을 만들어보세요 (난이도 : 下)

문제 2

a.txt 에 어떠한 긴 글이 들어 있는데, 이 글을 입력 받아서 특정한 문자열을 검색하는 프로그램을 만들어보세요 (난이도 : 中)

문제 3

a.txt 에 문자열을 입력 받아서 b.txt 에 그 문자열을 역으로 출력하는 프로그램을 만들어보세요 (난이도 : 中下)

파일 안에서 이동하기

안녕하세요~ 여러분. 파일 입출력의 관한 두 번째 강좌 입니다! 사실 지난번에는 뭔가 아쉬움이 남게 끝냈었습니다. 파일 입출력으로 무언가 제대로된 프로그램도 만들어 보지 않고 단순히 어떻게 사용하는지에 대해서만 간단히 다루어 보았었는데 이번에는 본격적으로 파일 입출력을 이용해서 무언가를 해보도록 하겠습니다.

파일 위치 지시자(File Position Indicator)

지난번에 파일 위치 지시자에 관해서 대충 설명하고 나갔는데요, 지난번의 설명이 무언가 부족하다는 느낌이 강하게 들어서 여기서 다시 한번 짚고 넘어가도록 하겠습니다. 스트림의 기본 모토는 바로 '순차적으로 입력을 받는다' 입니다.

즉 스트림에서 데이터를 입력 받을 때 에는 질서 정연하게 앞에 있는 데이터 먼저 순서대로 읽어들이게 되죠. 뒤에서 부터 거꾸로 읽는다나 데이터들을 뛰어 넘으며 읽어 들인다는 듯한 비정상적인 짓들을 하지 않습니다. 이렇게 순차적으로 읽어들이는 것을 가능하게 해주는 것이 바로 '파일 위치 지시자' 때문입니다.

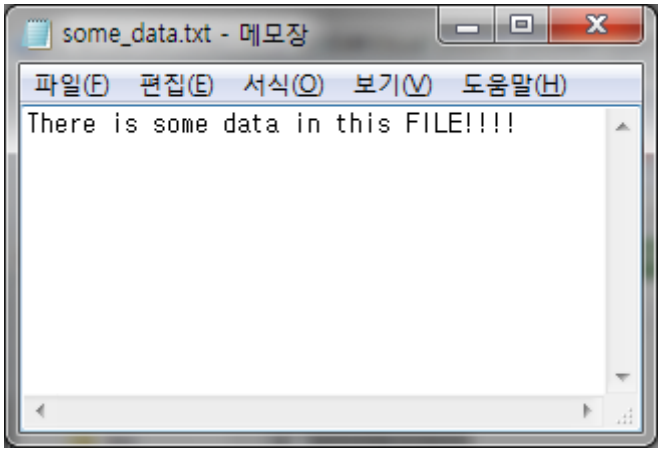
```c
/* 파일에서 문자를 하나씩 입력 받는다 */
#include <stdio.h>

int main() {
  FILE *fp = fopen("some_data.txt", "r");
  char c;

  if (fp == NULL) {
    printf("file open error ! \n");
    return 0;
  }

  while ((c = fgetc(fp)) != EOF) {
    printf("%c", c);
  }
}
```

성공적으로 컴파일 하였다면

실행 결과

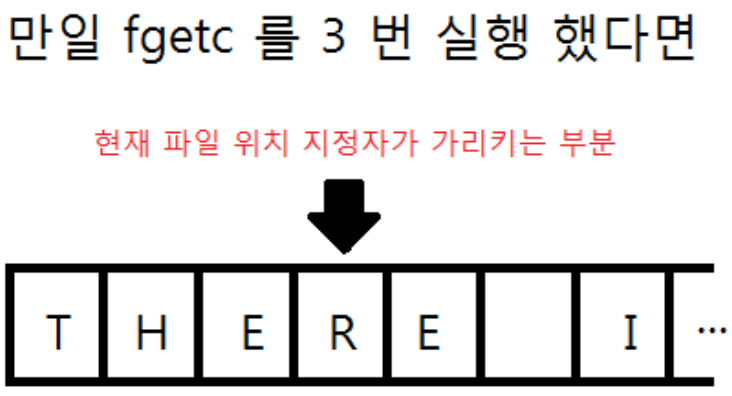
```
There is some data in this FILE!!!!
```

참고로 아래는 some_data.txt 에 들어 있었던 내용입니다. 당연하게도 화면에 출력된 내용과 정확히 일치합니다.



위 소스 코드에서 흥미로왔던 점은 없나요? 사실 이전 강좌에서도 다루었던 예제이지만 무언가 궁금한 점이 있을 것입니다. 바로 fgetc(fp) 를 실행할 때 마다 파일에 그 다음 문자를 입력 받는다는 점이지요. 컴퓨터는 어떻게 어디까지 읽어들었는지를 알고 이전에 입력 받았던 문자 바로 다음 문자를 입력 받는 것일까요? 그 이유는 간단합니다. 다음에 입력 받을 위치를 미리 표시해 놓으면 되지요. 이렇게 **다음에 입력 받아야 할 위치** 를 기억해 놓은 것을 **파일 위치 지시자** 가 하는 일입니다. 파일 위치 지시자는 파일에서 다음에 입력 받을 부분의 위치를 가리키고 있습니다.

예를 들어 위 예제에서 fgetc 를 세 번 호출했다고 해봅시다. 그렇다면 파일 위치 지시자는 아래 그림과 같은 위치를 가리키고 있게 됩니다.



어때요? 간단하지요. fgetc 를 호출하기 전에는 T 를 가리켰다가 한 번 호출하면 H, 두 번 호출하면 E, 세번 호출하면 R 을 가리키게 되지요. 따라서 다음번 fgetc 호출에서는 R 을 읽어들이고 파일 위치 지시자를 한 칸 옆으로 이동시킵니다.

이렇게 파일 위치 지시자가 다음으로 한 칸 씩 움직이는 바람에 데이터를 순차적으로 읽어들일 수

있게 되는 것이지요. 하지만 놀랍게도 C 언어에서는 파일 위치 지시자의 위치를 사용자 마음대로 바꿀 수 있게 해주는 여러가지 함수들을 지원해주고 있습니다. 그 중 대표적으로 가장 많이 사용하는 `fseek` 함수가 있지요. `fseek` 함수는 다음과 같이 생겼습니다. (자세한 설명은 C 언어 함수 레퍼런스 - fseek 함수를 참조하세요)

int fseek(FILE* stream, long int offset, int origin);

여기서 `stream` 에는 우리가 파일 위치 지시자를 옮기고 싶은 스트림의 포인터를, `origin` 에는 어디서 부터 옮길지, 그리고 `offset` 에는 얼마만큼 옮길지에 대한 정보가 들어가게 됩니다. 참고로 `origin` 에는 `SEEK_SET, SEEK_CUR, SEEK_END` 들이 있는데 각각 파일의 시작, 현재 파일 위치 지정자의 위치, 파일의 끝을 의미합니다. 그리고 `offset` 에는 `origin` 으로 부터 얼마나 옮길 것인지 숫자를 써주면 됩니다. 재미있는 점은 + 값을 쓰면 오른쪽으로 (위의 사진을 기준으로), - 값을 쓰면 왼쪽으로 파일 위치 지정자가 움직입니다.

```c
/* fseek 함수 예제 */
#include <stdio.h>
int main() {
  FILE *fp = fopen("some_data.txt", "r");
  char data[10];
  char c;

  if (fp == NULL) {
    printf("file open error ! \n");
    return 0;
  }

  fgets(data, 5, fp);
  printf("입력 받은 데이터 : %s \n", data);

  c = fgetc(fp);
  printf("그 다음에 입력 받은 문자 : %c \n", c);

  fseek(fp, -1, SEEK_CUR);

  c = fgetc(fp);
  printf("그렇다면 무슨 문자가? : %c \n", c);

  fclose(fp);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
입력 받은 데이터 : Ther
그 다음에 입력 받은 문자 : e
```

```
그렇다면 무슨 문자가? : e
```

와 같이 나오는 것을 볼 수 있습니다. 참고로, some_data.txt 에 있었던 내용은 이전 내용과 동일하게 There is some data in this FILE!!!! 입니다. 소스를 살펴보자면,

```
fgets(data, 5, fp);
```

위와 같이 fgets 함수를 통해서 fp 로 부터 입력을 받습니다. 이 때, 문자열 형태로 입력을 받는데 입력을 받을 때 \n 이 나올 때 까지 입력을 받거나 (두번째 인자의 크기 - 1) 만큼 입력을 받을 때 까지 입력을 받게 됩니다. 위 경우 \n 이 나오기 전에 4 바이트 만큼 입력을 받으므로 data 에는 *Ther* 이란 내용의 문자열이 들어갑니다. 참고적으로 왜 1 만큼 뺀 크기로 입력을 받냐면, data 에 문자열을 구성하기 위해 맨 뒤에는 언제나 NULL 문자를 위한 자리를 만들어주어야 하기 때문이죠.

이렇게 입력을 받게 된다면 이제 파일 위치 지정자는 e 를 가리키게 됩니다.

```
c = fgetc(fp);
printf("그 다음에 입력 받은 문자 : %c \n", c);
```

그 다음에 fgetc 로 fp 에서 문자를 받아 오면 역시 생각했던 대로 e 가 출력이 되겠지요. 그리고 파일 위치 지정자는 다시 한 칸 옆으로 옮겨가서 ' ' 을 가리키게 됩니다. 띄어쓰기도 엄연한 문자이지요. (즉, 띄어쓰기에도 ASCII 값이 당연히 대응되어 있습니다)

```
fseek(fp, -1, SEEK_CUR);
```

드디어 fseek 함수를 사용했습니다. 앞에서 말했듯이 SEEK_CUR 은 현재 파일 위치 지정자의 위치를 나타내고, 두번째 인자로 -1 을 전달했으므로 왼쪽으로 1 만큼 옮기라는 것이지요. 즉, 현재 파일 위치 지정자의 위치에서 왼쪽으로 1 만큼 다시 옮겼으니 ' ' 을 가리키고 있던 파일 위치 지정자가 이전의 'e' 를 가리키게 됩니다. 따라서 다시

```
c = fgetc(fp);
printf("그렇다면 무슨 문자가? : %c \n", c);
```

을 호출하여 문자를 입력 받으면 다시 e 가 나오게 되는 것이지요. 어때요. 간단하지요?

```
/* 파일의 마지막 문자를 보기*/
#include <stdio.h>

int main() {
  FILE *fp = fopen("some_data.txt", "r");
  char data[10];
```

```
  char c;

  if (fp == NULL) {
    printf("file open error ! \n");
    return 0;
  }

  fseek(fp, -1, SEEK_END);
  c = fgetc(fp);
  printf("파일 마지막 문자 : %c \n", c);

  fclose(fp);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
파일 마지막 문자 : !
```

와 같이 잘 나옵니다. 참고로 some_data.txt 에는 여태까지 위에서 써왔던 데이터인 There is some data in this FILE!!!! 가 들어있습니다. 이 때 이 파일의 마지막 문자는 ! 가 되겠지요.

```
fseek(fp, -1, SEEK_END);
```

위 소스에서 가장 중요한 부분은 딱 위 하나 입니다. 파일 위치 지정자를 파일의 맨 끝에서 한 칸 왼쪽으로 간 부분으로 옮깁니다. 왜 한 칸 왼쪽으로 옮기냐면, 맨 끝으로 옮기게 되면 그 부분에는 EOF (파일의 끝) 을 나타내는 것이 들어 있기 때문에 우리가 원하는 결과가 아니게 되지요. 우리가 파일에 입력한 맨 마지막 문자는 EOF 바로 왼쪽에 위치한 ! 가 됩니다.

파일에 쓰기, 읽기 같이 하기

여태까지 여러분은 하나의 파일에 읽기 또는 쓰기 작업을 한 번에 하나씩 밖에 할 수 없었습니다. 그런데 다행스럽게도 fopen 에는 하나의 파일에 대해 읽기/쓰기를 모두 할 수 있는 방법을 지원해 줍니다.

```
/* fopen 의 "r+" 인자 이용해보기 */
#include <stdio.h>
int main() {
  FILE *fp = fopen("some_data.txt", "r+");
  char data[100];
```

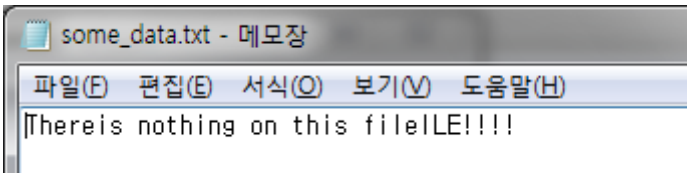
```
  fgets(data, 100, fp);
  printf("현재 파일에 있는 내용 : %s \n", data);

  fseek(fp, 5, SEEK_SET);

  fputs("is nothing on this file", fp);

  fclose(fp);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
There is some data in this FILE!!!!
```

와 같이 잘 나옵니다. 그리고 수정된 some_data.txt 의 모습은 아래와 같습니다.



아주 잘 되는군요. 일단 맨 처음에

```
FILE *fp = fopen("some_data.txt", "r+");
```

같이 하였습니다. 이는 some_data.txt 를 **읽기 및 쓰기형식으로 열겠다** 라는 뜻인데, 파일이 존재하지 않는다면 열지를 않겠다는 의미 입니다. 만일 파일이 존재한다면 파일의 내용을 지우지 않지요. 반면에 뒤에서 배울 "w+" 도 "읽기 및 쓰기 형식으로 열겠다" 이지만, 이 경우 파일이 존재하지 않는다면 파일을 새로 만들고 파일이 존재한다면 파일의 내용을 싹 지워버리게 됩니다.

```
fgets(data, 100, fp);
printf("현재 파일에 있는 내용 : %s \n", data);
```

를 통해 파일의 있는 내용들을 모두 읽어들였습니다 (정확히 말하면 최대 100 바이트 까지 읽지만 우리의 파일의 내용은 100 바이트 보다 작으므로 모두 읽어들여다고 보면 됩니다). 그리고 이와 함께 파일 위치 지정자도 파일 맨 끝을 가리키고 있겠지요.

```
fseek(fp, 5, SEEK_SET);
```

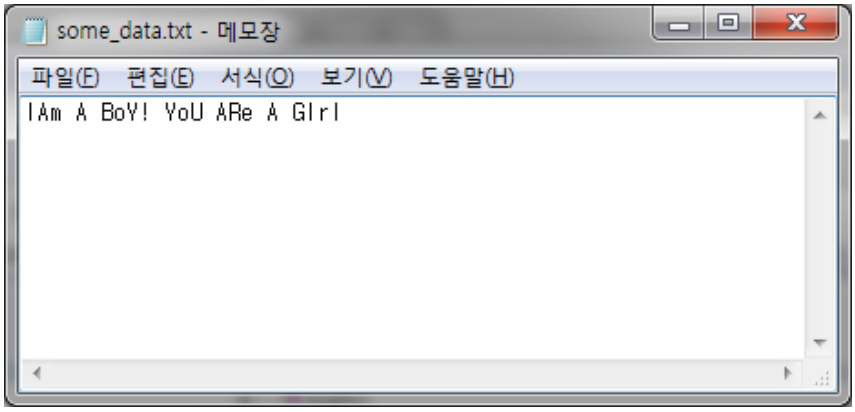
이에 위와 같이 fseek 함수를 이용하여 파일의 맨 앞에서 5 칸 떨어진 곳으로 이동해봅시다. 0 칸 떨어졌을 때는 T, 1 칸은 h, ... 와 같은 방식으로 세보면 5 칸 떨어진 곳은 ' ' 임을 알 수 있습니다. (공백 문자가 위치한 곳) 그리고 이제 여기에

```
fputs("is nothing on this file", fp);
```

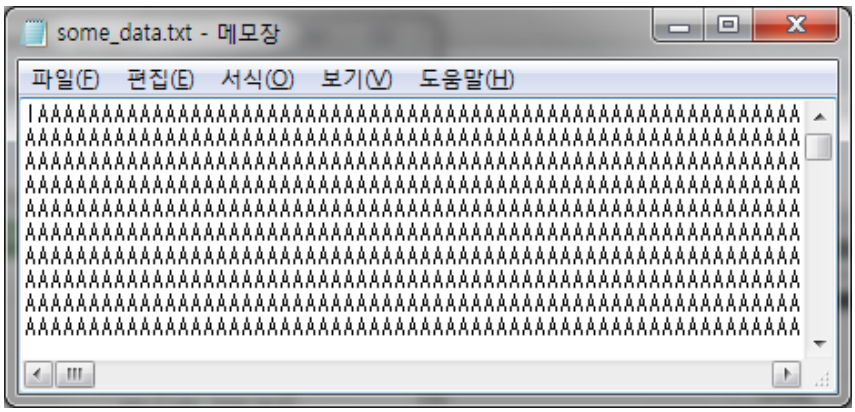
를 하게 되면 이전에 있던 내용은 무시하고 is nohting on this file 이 차지하는 만큼 덮어 씌우기가 됩니다. 따라서 위의 사진에서도 나타나듯이 파일에 위와 같이 나타나게 되지요.

```
/* 특정한 파일을 입력 받아서 소문자를 대문자로, 대문자를 소문자로 바꾸는
 * 프로그램*/
#include <stdio.h>
int main() {
  FILE *fp = fopen("some_data.txt", "r+");
  char c;

  if (fp == NULL) {
    printf("파일 열기를 실패하였습니다! \n");
    return 0;
  }

  while ((c = fgetc(fp)) != EOF) {
    /* c 가 대문자일 경우 */
    if (65 <= c && c <= 90) {
      /* 한 칸 뒤로 가서*/
      fseek(fp, -1, SEEK_CUR);
      /* 소문자로 바뀐 c 를 출력한다*/
      fputc(c + 32, fp);

    }
    /* c 가 소문자일 경우*/
    else if (97 <= c && c <= 122) {
      fseek(fp, -1, SEEK_CUR);
      fputc(c - 32, fp);
    }
  }

  fclose(fp);
}
```

성공적으로 컴파일 하였다면 원래는 아래와 같은 파일이



와 같이 괴상하게 변해버렸습니다.



사실 프로그램이 종료 되지도 않아서 강제로 종료해야만 했었습니다.

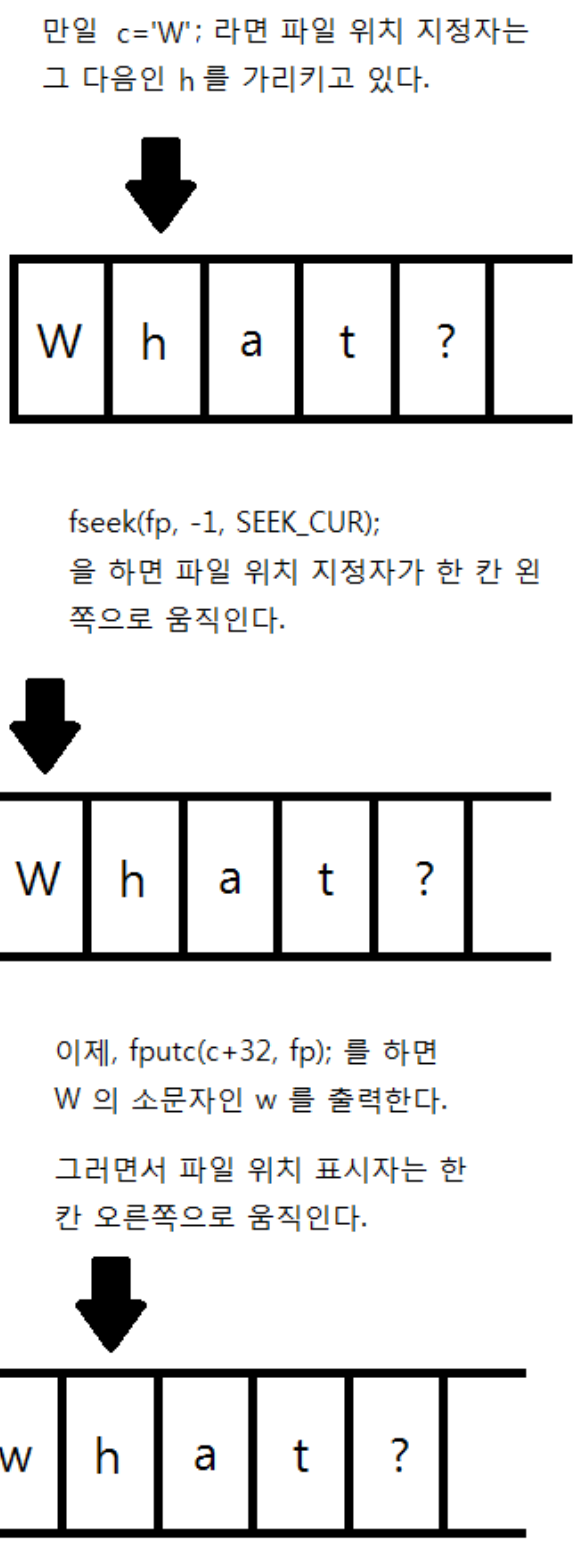
도대체 왜 이런 일이 발생한 것일까요?

```c
while ((c = fgetc(fp)) != EOF) {
  /* c 가 대문자일 경우 */
  if (65 <= c && c <= 90) {
    /* 한 칸 뒤로 가서*/
    fseek(fp, -1, SEEK_CUR);
    /* 소문자로 바뀐 c 를 출력한다*/
    fputc(c + 32, fp);

  }
  /* c 가 소문자일 경우*/
  else if (97 <= c && c <= 122) {
    fseek(fp, -1, SEEK_CUR);
    fputc(c - 32, fp);
  }
```

위 소스를 보면 큰 문제는 없어 보입니다. 일단 대문자일 경우만 살펴보면 ASCII 표를 보면 영어 대문자의 경우 값이 65 ~ 90 에 있으므로 위와 같이 if 문을 설정하면 대문자들을 처리할 수 있습니다.

```
/* 한 칸 뒤로 가서*/
fseek(fp, -1, SEEK_CUR);
/* 소문자로 바뀐 c 를 출력한다*/
fputc(c + 32, fp);
```

사실 위 과정에서는 문제가 없습니다.예를 들어서 What? 이란 문자열이 있을 때 c 에 W 가 들어있다면 현재 파일 위치 지시자는 그 다음인 h 를 가리키고 있을 것입니다. 따라서 w 부분에 W 를 쓰기 위해 파일 위치 지시자를 한 칸 뒤로 옮겨서 w 를 가리키게 하고, fputc 를 통해 (c 에 32 를 더한 값, 아스키 코드표를 보면 그 대문자에 해당하는 소문자값 임을 알 수 있다) 을 써서 결과적으로 what? 가 됩니다.


만일 c='W'; 라면 파일 위치 지정자는
그 다음인 h 를 가리키고 있다.
W h a t ?
fseek(fp, -1, SEEK_CUR);
을 하면 파일 위치 지정자가 한 칸 왼
쪽으로 움직인다.
W h a t ?
이제, fputc(c+32, fp); 를 하면
W 의 소문자인 w 를 출력한다.
그러면서 파일 위치 표시자는 한
칸 오른쪽으로 움직인다.
w h a t ?

그런데 도대체 무엇이 문제일까요? 사실 그 이유는 간단합니다.

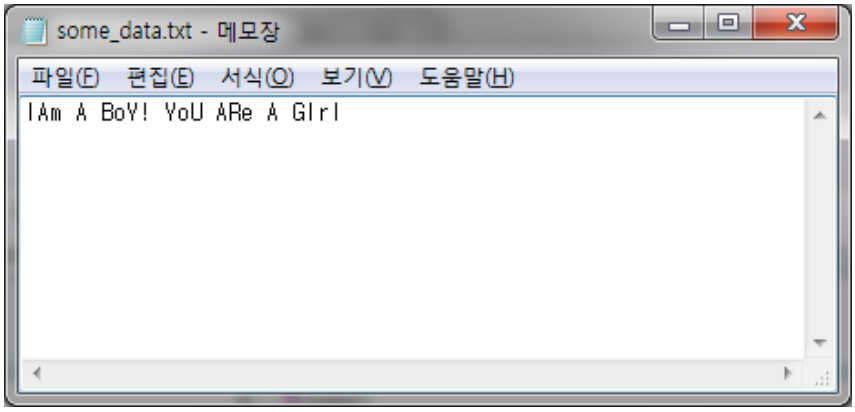
스트림 작업에서 읽기/쓰기 를 변환할 때 에는 반드시 `fflush` 함수를 호출하거나 `fseek` 이나 `rewind` 와 같은 함수를 호출하여 파일 위치 지정자를 다시 설정해주어야 하기 때문(자세한 내용은

fopen 함수 레퍼런스를 참조)입니다. 따라서 반드시 위와 같이 쓰기 작업 후 다시 읽기 작업 (while 문에서 fputc 를 통해 읽기 작업이 수행된다) 을 할 때 에는 `fflush` 나 `fseek` 함수를 호출해 주시기 바랍니다.

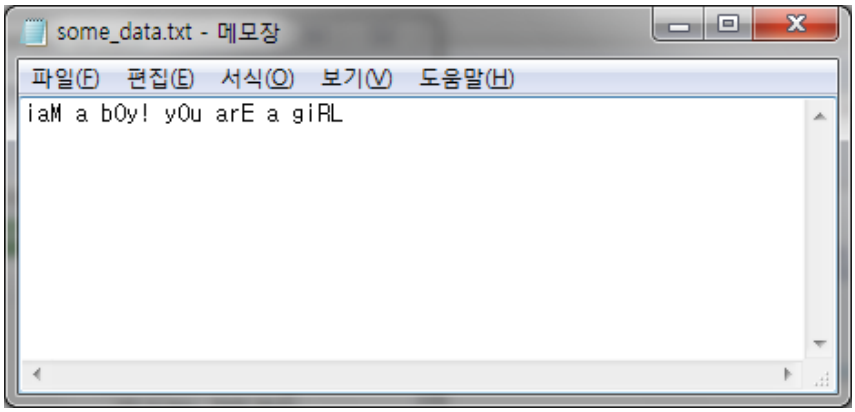
이를 토대로 코드를 수정해보았습니다.

```c
#include <stdio.h>

int main() {
  FILE *fp = fopen("some_data.txt", "r+");
  char c;

  if (fp == NULL) {
    printf("파일 열기를 실패하였습니다! \n");
    return 0;
  }

  while ((c = fgetc(fp)) != EOF) {
    /* c 가 대문자일 경우 */
    if (65 <= c && c <= 90) {
      /* 한 칸 뒤로 가서*/
      fseek(fp, -1, SEEK_CUR);
      /* 소문자로 바뀐 c 를 출력한다*/
      fputc(c + 32, fp);
      /*

      쓰기 - 읽기 모드 전환을 위해서는 무조건
      fseek 함수와 같은 파일 위치 지정자 설정 함수들을
      호출해야 한다.

      */
      fseek(fp, 0, SEEK_CUR);
    }
    /* c 가 소문자일 경우*/
    else if (97 <= c && c <= 122) {
      fseek(fp, -1, SEEK_CUR);
      fputc(c - 32, fp);
      fseek(fp, 0, SEEK_CUR);
    }
  }

  fclose(fp);
}
```

성공적으로 컴파일 하였다면



위와 같았던 파일 내용이



아래와 같이 예쁘게 바뀝니다.

```
/*

쓰기 - 읽기 모드 전환을 위해서는 무조건
fseek 함수와 같은 파일 위치 지정자 설정 함수들을
호출해야 한다.

*/
fseek(fp, 0, SEEK_CUR);
```

위 소스에서 굳이 파일 위치 지정자 의 위치를 옮길 필요가 없음에도 불구하고 fseek 함수를 통해 파일 위치 지정자를 설정하였습니다. 사실 위와 같이 fseek 함수를 호출하면 파일 위치 지정자는 하나도 옮겨지지 않습니다. 단순히 쓰기작업에서 읽기 작업으로 바꾸기 위해 fseek 함수를 호출한 것 뿐이지요.

만일 위의 fseek 이 마음에 들지 않는다면

```
/* 한 칸 뒤로 가서*/
fseek(fp, -1, SEEK_CUR);
```

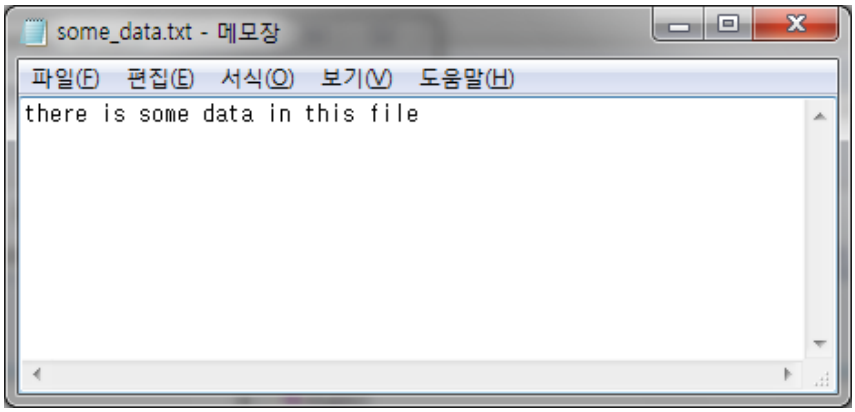
```
/* 소문자로 바뀐 c 를 출력한다*/
fputc(c + 32, fp);

fflush(fp);
```

로 하셔도 됩니다. 아무튼 `fseek` 이든 `fflush` 함수든 호출해 주어야만 합니다.

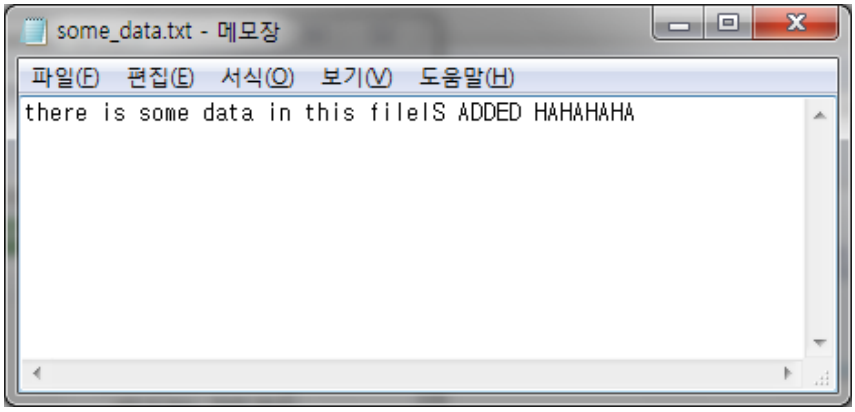
fopen 함수의 기타 인자 사용

```c
/* fopen 의 'append' 기능 사용*/
#include <stdio.h>
int main() {
  FILE *fp = fopen("some_data.txt", "a");
  char c;
  if (fp == NULL) {
    printf("파일 열기를 실패하였습니다! \n");
    return 0;
  }
  /* 아래 내용이 파일 뒤에 덧붙여진다.*/
  fputs("IS ADDED HAHAHAHA", fp);
  fclose(fp);
}
```

성공적으로 컴파일 하였다면 아래와 같았던 파일 내용이



아래 처럼 바뀝니다.



fopen 부분을 살펴보면

```
FILE *fp = fopen("some_data.txt", "a");
```

로 파일을 "a" 형식을 열었습니다. 이 뜻은 파일을 **덧붙이기(append)** 형식으로 연다는 의미입니다. 기존의 "w" 로 열었을 때에는 파일의 내용이 모두 지워지는 대신에 맨 앞부터 내용이 쓰여졌는데 덧붙이기 형식에서는 파일의 맨 끝부분 부터 내용이 쓰여지고 앞 부분은 전혀 건들여지지 않습니다. 즉, 이전에 파일에 들어가 있었던 내용들은 아주 소중하게 보호가 됩니다.

"r+" 나 "w+" 와 마찬가지로 "a+" 형식도 있는데 이도 마찬가지로 읽기/덧붙이기 를 번갈아가면서 할 수 있습니다. 참고로 읽는 작업은 파일 어디에서든지 사용할 수 있지만 쓰기 작업의 경우 아무리 파일 위치 지시자를 이동 시켜 보아도 기존파일의 끝 부분 위치에서 부터만 쓸 수 있습니다.

fscanf 사용하기

```
/* fscanf 이용하기 */
#include <stdio.h>

int main() {
  FILE *fp = fopen("some_data.txt", "r");
  char data[100];

  if (fp == NULL) {
    printf("파일 열기 오류! \n");
    return 0;
  }

  printf("---- 입력 받은 단어들 ---- \n");

  while (fscanf(fp, "%s", data) != EOF) {
    printf("%s \n", data);
  }
```

```
  fclose(fp);
}
```

성공적으로 컴파일 하였다면

실행 결과

```
---- 입력 받은 단어들 ----
There
is
some
data
in
this
FILE!!!!
```

와 같이 잘 나옵니다. fscanf 함수는 우리가 여태까지 사용해왔던 scanf 함수와 아주 아주 유사한데, 사실 scanf 가 stdin 에서만 입력을 받고 fscanf 는 임의의 스트림에서도 입력을 받을 수 있는 좀더 일반화 된 함수라고 보시면 됩니다.

fscanf 함수의 첫번째 인자로 입력을 받을 스트림을 써주게 되는데, 따라서

```
fscanf(stdin, "%s", data);
scanf("%s", data);
```

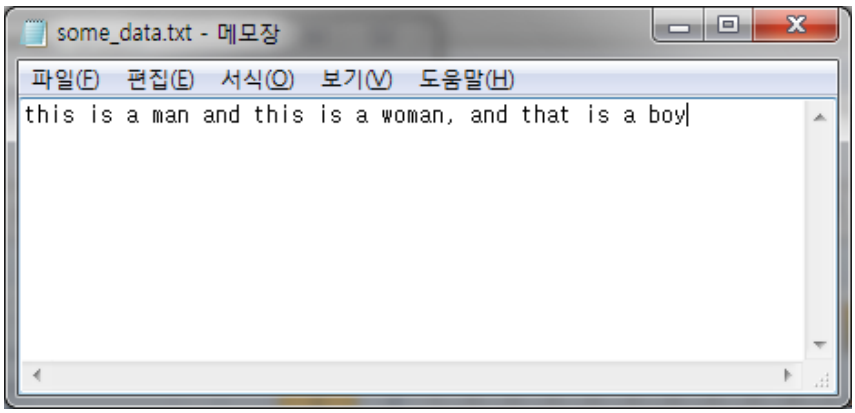
는 정확히 일치하는 문장들 입니다. 아무튼 fscanf 는 사용자가 지정한 형식에 맞게 데이터를 읽어오게 되는데 fgets 와는 달리 띄어쓰기나 탭 문자들도 모두 인식하므로 위와 같이 각각의 단어들을 읽어오는데 요긴하게 사용할 수 있습니다.

```
while (fscanf(fp, "%s", data) != EOF) {
  printf("%s \n", data);
}
```

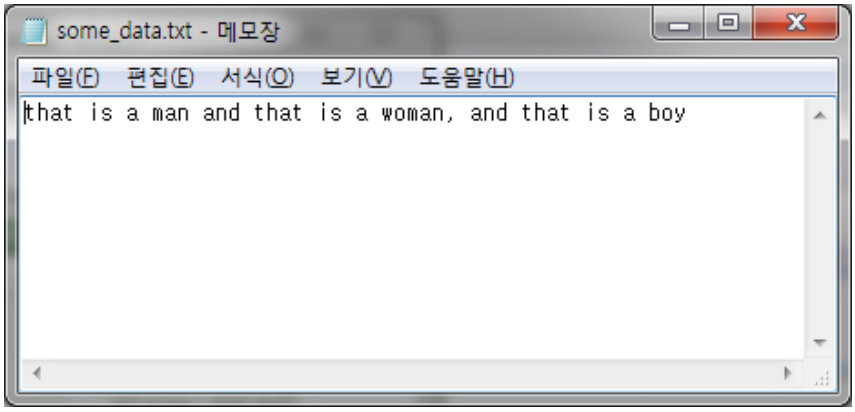
일단 위 소스에서 가장 중요한 부분을 봅시다. fscanf 를 통해 fp 에서 문자열을 읽어오고 있는데 fgets 는 \n 이 나올 때 까지 하나의 문자열로 보고 받아들이지만 fscanf 는 띄어쓰기나 탭 문자 (\t), \n 들 중 어느 하나가 나올 때 까지 입력 받으므로 파일에서는 각 단어들을 하나씩 읽어들이게 됩니다. 물론 읽어 들인 만큼 파일 위치 지정자는 이동하게 되지요. 이 때 fscanf 가 더이상 새로운 데이터를 입력을 받을 수 없을 경우에는 EOF 를 리턴하게 됩니다. 즉, 파일의 끝에 도달하면 EOF 를 리턴하여 while 문을 빠져나갑니다.

```c
/* 파일에서 'this' 를 'that' 으로 바꾸기*/
#include <stdio.h>
#include <string.h>

int main() {
  FILE *fp = fopen("some_data.txt", "r+");
  char data[100];

  if (fp == NULL) {
    printf("파일 열기 오류! \n");
    return 0;
  }

  while (fscanf(fp, "%s", data) != EOF) {
    if (strcmp(data, "this") == 0) {
      fseek(fp, -(long)strlen("this"), SEEK_CUR);
      fputs("that", fp);

      fflush(fp);
    }
  }

  fclose(fp);
}
```

성공적으로 컴파일 하였다면



위와 같았던 파일이



아래 처럼 this 들이 모두 that 으로 바뀐 것을 보실 수 있습니다. 사실 그 원리는 아주 간단합니다.

```
while (fscanf(fp, "%s", data) != EOF) {
  if (strcmp(data, "this") == 0)
```

이전 예제에서와 같은 방식으로 fscanf 들을 통해 파일에서 단어들을 입력받는데, 각각의 단어들을 strcmp 함수를 이용하여 this 와 같은지 비교를 하지요. 만일 같다면 이제 this 를 that 으로 덮어 씌우기만 하면 됩니다.

```
fseek(fp, -(long)strlen("this"), SEEK_CUR);
fputs("that", fp);
```

fscanf 에서 “this” 를 입력 받은 시점에서 파일 위치 지정자는 this 바로 다음 문자를 가리키고 있으므로 “this” 의 길이만큼 왼쪽으로 이동시킨다면 파일 위치 지정자는 t 를 가리키게 되지요. 이제 이 상태에서 fputs 로 “that” 을 쓴다면 “this” 가 들어가 있던 자리에 “that” 이 정확히 자리를 대체하게 됩니다. 그리고 마지막으로

```
fflush(fp);
```

위 쓰기 작업이 끝나면 다시 while 문에서 fscanf 로 읽기 작업을 하게 되므로 fflush 를 사용해 주어야만 합니다. 물론 이전 처럼 fseek 를 사용하셔도 되고요

파일 입출력 실제로 적용해보기

이제 본격적으로 파일 입출력을 이용해서 무언가를 해보아야 겠죠?가장 먼저 여태가지 만들어보았던 도서 관리 프로그램에 입출력 기능을 적용시켜봅시다.

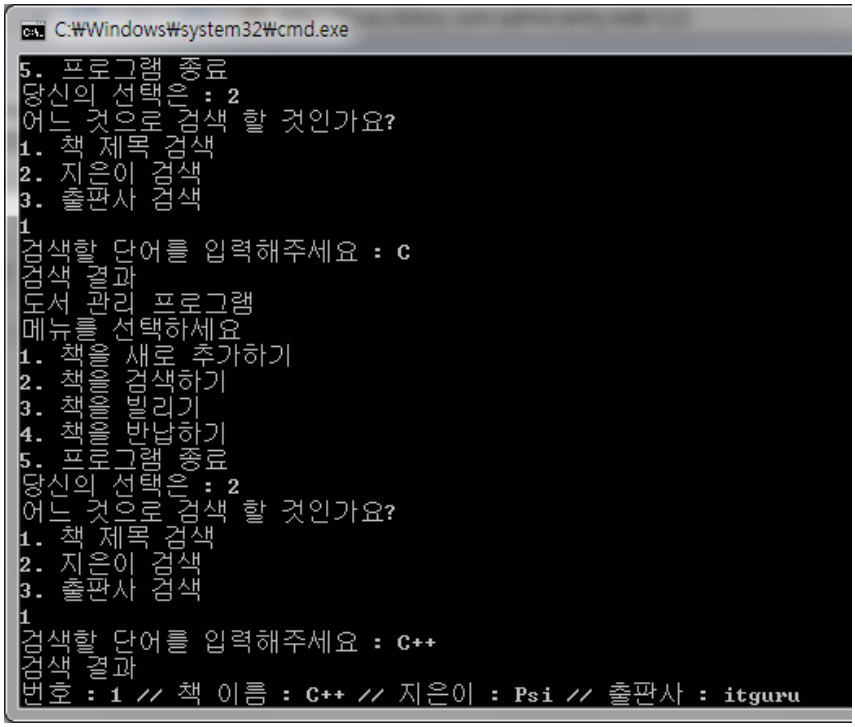
일단 아래는 입출력 기능을 적용시키기 전 단계의 도서 관리 프로그램으로 여태까지 배운 새로운 기술들을 이용하여 작성하였습니다.

```c
/*
지난번에 만들었었던 도서 관리 프로그램으로 우리가 여태까지 배운 최신 C 언어
기술들(!!) 을 이용하여 새롭게 만들었습니다. 어떻게 보면
http://itguru.tistory.com/60 의 생각해 볼 문제의 두번째 문제의 해답이 되기도
하겠군요*/
#include <stdio.h>
#include <stdlib.h>

struct BOOK {
  char book_name[30];
  char auth_name[30];
  char publ_name[30];
  int borrowed;
};
typedef struct BOOK BOOK;

char compare(char *str1, char *str2);
int register_book(BOOK 02 * book_list, int *nth);
int search_book(BOOK *book_list, int total_num_book);
int borrow_book(BOOK *book_list);
int return_book(BOOK *book_list);

int main() {
  int user_choice;        /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */
  BOOK *book_list;
  printf("도서관의 최대 보관 장서 수를 설정해주세요 : ");
  scanf("%d", &user_choice);
  book_list = (BOOK *)malloc(sizeof(BOOK) * user_choice);
  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");
    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);
    if (user_choice == 1) { /* 책을 새로 추가하는 함수 호출 */
      register_book(book_list, &num_total_book);
    } else if (user_choice == 2) { /* 책을 검색하는 함수 호출 */
      search_book(book_list, num_total_book);
    } else if (user_choice == 3) { /* 책을 빌리는 함수 호출 */
      borrow_book(book_list);
    } else if (user_choice == 4) { /* 책을 반납하는 함수 호출 */
      return_book(book_list);
    } else if (user_choice == 5) { /* 프로그램을 종료한다. */
      break;
    }
```

```
  }
  free(book_list);
  return 0;
} /* 책을 추가하는 함수*/
int register_book(BOOK *book_list, int *nth) {
  printf("책의 이름 : ");
  scanf("%s", book_list[*nth].book_name);
  printf("책의 저자 : ");
  scanf("%s", book_list[*nth].auth_name);
  printf("책의 출판사 : ");
  scanf("%s", book_list[*nth].publ_name);
  book_list[*nth].borrowed = 0;
  (*nth)++;
  return 0;
} /* 책을 검색하는 함수 */
int search_book(BOOK *book_list, int total_num_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];
  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);
  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);
  printf("검색 결과 \n");
  if (user_input == 1) {
    /* i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을 사용자가 입력한
     * 검색어와 비교하고 있다. */
    for (i = 0; i < total_num_book; i++) {
      if (compare(book_list[i].book_name, user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_list[i].book_name, book_list[i].auth_name,
               book_list[i].publ_name);
      }
    }
  } else if (user_input == 2) {
    /* i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을 사용자가
     * 입력한 검색어와 비교하고 있다. */
    for (i = 0; i < total_num_book; i++) {
      if (compare(book_list[i].auth_name, user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_list[i].book_name, book_list[i].auth_name,
               book_list[i].publ_name);
      }
    }
  } else if (user_input == 3) {
    /* i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를 사용자가 입력한
     * 검색어와 비교하고 있다. */
    for (i = 0; i < total_num_book; i++) {
```

```
      if (compare(book_list[i].publ_name, user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_list[i].book_name, book_list[i].auth_name,
               book_list[i].publ_name);
      }
    }
  }
  return 0;
}
char compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }
    str1++;
    str2++;
  }
  if (*str2 == '\0') return 1;
  return 0;
}
int borrow_book(BOOK *book_list) { /* 사용자로 부터 책 번호를 받을 변수 */
  int book_num;
  printf("빌릴 책의 번호를 말해주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &book_num);
  if (book_list[book_num].borrowed == 1) {
    printf("이미 대출된 책입니다! \n");
  } else {
    printf("책이 성공적으로 대출되었습니다. \n");
    book_list[book_num].borrowed = 1;
  }
  return 0;
}
int return_book(BOOK *book_list) { /* 반납할 책의 번호 */
  int num_book;
  printf("반납할 책의 번호를 써주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &num_book);
  if (book_list[num_book].borrowed == 0) {
    printf("이미 반납되어 있는 상태입니다\n");
  } else {
    book_list[num_book].borrowed = 0;
    printf("성공적으로 반납되었습니다\n");
  }
  return 0;
}
```

성공적으로 컴파일 하였다면



와 같이 여러가지 재미있는 것들을 할 수 있습니다.

위 소스에서는 여태까지 배운 것들을 대부분 사용하였는데요, 예를 들면

```
struct BOOK {
  char book_name[30];
  char auth_name[30];
  char publ_name[30];
  int borrowed;
};

typedef struct BOOK BOOK;
```

typedef 를 통해서 귀찮게 struct BOOK 이라고 매번 써야 하는 대신에 BOOK 이라고 해도 struct BOOK 의 의미를 지니게 하였습니다. 참고적으로 위 문장들은

```
typedef struct BOOK {
  char book_name[30];
  char auth_name[30];
  char publ_name[30];
  int borrowed;
} BOOK;
```

와 같이 써도 동일한 의미를 지닙니다.

또한 재미있는 부분으로 다음과 같이 동적할당을 이용하였는데

```
printf("도서관의 최대 보관 장서 수를 설정해주세요 : ");
scanf("%d", &user_choice);

book_list = (BOOK *)malloc(sizeof(BOOK) * user_choice);
```

위와 같이 하여서 book_list 의 책들의 최대 보관 개수를 지정하여 그 크기에 딱 맞는 배열을 생성하도록 하였습니다. 이전과 함수의 모습도 많이 달라졌는데

```
int add_book(char (*book_name)[30], char (*auth_name)[30],
             char (*publ_name)[30], int *borrowed, int *num_total_book);
int search_book(char (*book_name)[30], char (*auth_name)[30],
                char (*publ_name)[30], int num_total_book);
int borrow_book(int *borrowed);
int return_book(int *borrowed);
```

가 기존의 구조체를 쓰지 않았을 때의 함수들의 모습이라면 아래는

```
int register_book(BOOK *book_list, int *nth);
int search_book(BOOK *book_list, int total_num_book);
int borrow_book(BOOK *book_list);
int return_book(BOOK *book_list);
```

구조체를 써서 훨씬 간단해진 함수들의 모습 입니다.

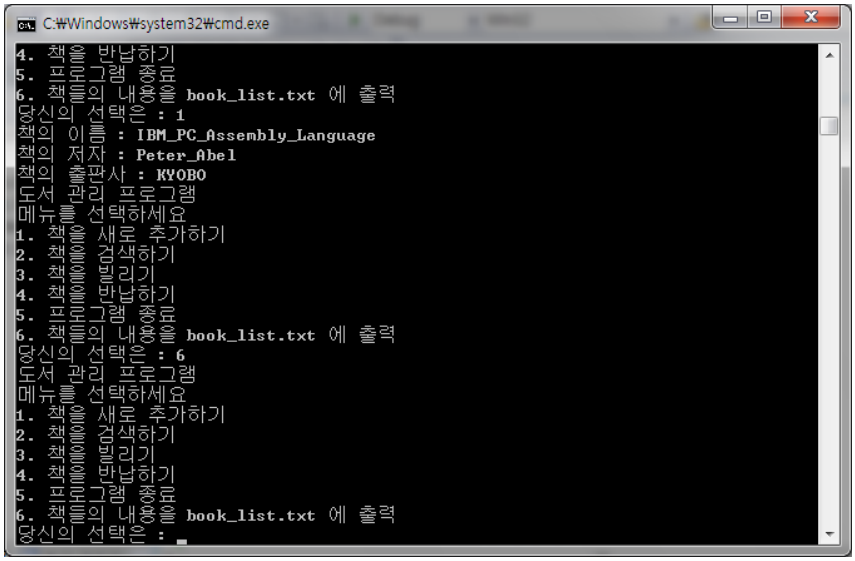
아무튼 위 소스에 대한 설명은 이정도로 마치도록 하고 (나머지 부분은 여러분이 스스로 분석/개량해보세요!) 이제 본격적으로 파일 입출력을 도서 관리 프로그램에 적용시켜 봅시다. 먼저 우리가 하고 싶은 일은 현재 도서관에 등록된 장서들의 목록을 예쁘게 파일에 출력시키는 것입니다. 사실 이는 간단하므로 여러분들도 만들어보시기 바랍니다.

```
int register_book(BOOK *book_list, int *nth);
int search_book(BOOK *book_list, int total_num_book);
int borrow_book(BOOK *book_list);
int return_book(BOOK *book_list);
int print_book_list(BOOK *book_list, int total_num_book);
int main() {
  int user_choice;        /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */

  BOOK *book_list;

  printf("도서관의 최대 보관 장서 수를 설정해주세요 : ");
  scanf("%d", &user_choice);

  book_list = (BOOK *)malloc(sizeof(BOOK) * user_choice);
```

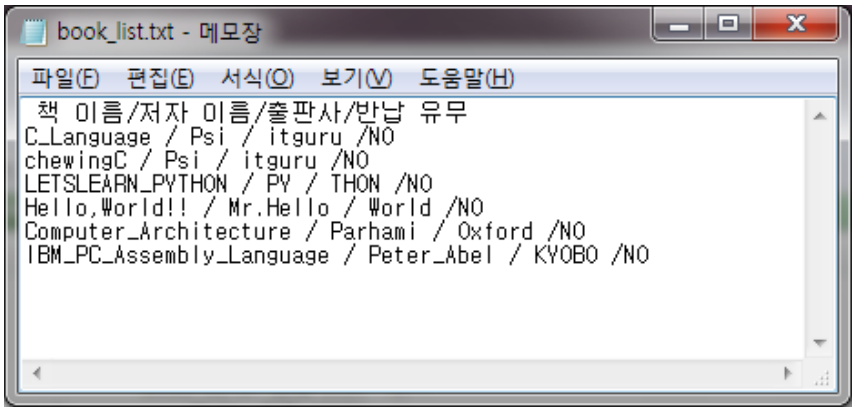
```
  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");
    printf("6. 책들의 내용을 book_list.txt 에 출력 \n");

    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);

    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
      register_book(book_list, &num_total_book);
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
      search_book(book_list, num_total_book);
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
      borrow_book(book_list);
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
      return_book(book_list);
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    } else if (user_choice == 6) {
      print_book_list(book_list, num_total_book);
    }
  }

  free(book_list);
  return 0;
}
int print_book_list(BOOK *book_list, int total_num_book) {
  FILE *fp = fopen("book_list.txt", "w");
  int i;

  if (fp == NULL) {
    printf("출력 오류 ! \n");
    return -1;
  }

  fprintf(fp, " 책 이름/저자 이름/출판사/반납 유무\n");
  for (i = 0; i < total_num_book; i++) {
    fprintf(fp, "%s / %s / %s", book_list[i].book_name, book_list[i].auth_name,
            book_list[i].publ_name);
    if (book_list[i].borrowed == 0)
      fprintf(fp, " /NO \n");
```

```
    else
      fprintf(fp, " /YES \n");
  }

  fclose(fp);
}
```

바뀐 부분만 보면 위와 같습니다.성공적으로 컴파일 하였다면



와 같이 나옵니다.이제 6 번을 눌러서 출력을 해보면



와 같이 아주 예쁘게 파일에 출력되었음을 알 수 있습니다.

위 소스 코드에서 주목해야 할 부분은 바로 파일에 내용을 출력하는 `print_book_list` 함수 입니다. `print_book_list` 에서 새로운 입출력 함수를 사용하였는데 바로 `fprintf` 함수 입니다. 이 함수는 `printf` 와 비슷하게 생겼는데 `printf` 의 경우 인자로 지정한 내용을 콘솔 화면(정확히 말하면 `stdout`) 에 출력하는 반면에 `fprintf` 함수는 지정한 스트림에 출력하게 되지요. 다시 말하면

```
fprintf(stdout, "Hello, World! \n");
printf("Hello, World! \n");
```

은 정확히 동일한 작업을 하게 됩니다. 아무튼, printf 가 화면에 출력한다면 fprintf 는 스트림에 출력하신다고 생각하면 됩니다. 그렇다면

```
fprintf(fp, " 책 이름/저자 이름/출판사/반납 유무\n");
for (i = 0; i < total_num_book; i++) {
  fprintf(fp, "%s / %s / %s", book_list[i].book_name, book_list[i].auth_name,
          book_list[i].publ_name);
  if (book_list[i].borrowed == 0)
    fprintf(fp, " /NO \n");
  else
    fprintf(fp, " /YES \n");
}
```

은 쉽게 이해할 수 있으리라 봅니다. 맨 첫번째 문장에서 책 이름/저자 이름/출판사/반납 유무를 화면에 출력했다면, 아래 for 문에서 book_list 에 들어있는 책의 정보들을 모두 표시하게 되지요. 상당히 간단하지요?

원래는 도서 관리 프로그램에서 출력한 데이터를 읽어들이는 작업도 같이 해볼려고 했는데 이 부분은 여러분들께 생각해보기로 남기겠습니다. 그럼 이번 강좌는 여기에서 마치도록 하겠고요, 다음 시간까지도 파일 입출력에 대해서 좀더 이야기 보도록 하겠습니다!

생각해보기

문제 1

위의 도서관리 프로그램에서 출력한 도서 목록을 입력 받아서 배열에 집어 넣는 작업을 만들어보세요. 참고로 도서 목록 출력 파일의 형식은 아래와 같다고 합시다.

> 전체 책의 개수 책 이름 저자 이름 출판사 이름 대출 유무 책 이름 저자 이름 출판사 이름 대출 유무

예를 들면

> 2 2 C언어 Psi Psi itguru offset AAA

와 같은 형식 이지요. 물론 파일에 도서 목록을 출력하는 작업도 수정해야 되겠지요. (난이도 : 中)

문제 2

파일에서 특정한 단어를 검색하여 몇 번째 줄에 나오는지 모두 출력하는 프로그램을 만드세요. (난이도 : 上)(참고로 1 줄의 기준은 \n 의 유무로 합시다. 따라서 `fscanf` 를 사용하면 안되겠지요?)

문제 3

파일에서 특정한 문자를 검색하여 몇 개나 나오는지 출력하는 프로그램을 만드세요. (난이도 : 下)

도서 관리 프로그램의 완성

안녕하세요 여러분! 이제 슬슬 파일 입출력도 마무리를 해가며 저의 C 언어 강좌도 끝을 향해 달려갑니다. 이번 강좌에서는 파일 입출력을 이용한 여러 가지 프로그램을 만들어보면서 파일 입출력에 대해 조금 더 친해지도록 하지요 :)

지난 강좌에서 여러분들과 함께 도서 관리 프로그램을 열심히 만들었었습니다. 그리고 제가 여러분들께 '생각해보기' 로 입력을 받는 형태로 바꿔보자고 이야기 했었죠. 여러분들은 모두 해보셨나요? 아직 오랜 시간 고민을 하지 안 하신 분들은 살포시 뒤로가기를 눌러주시기 바랍니다.

```c
/* 도서 관리 프로그램의 전체 소스 코드*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct BOOK {
  char book_name[30];
  char auth_name[30];
  char publ_name[30];
  int borrowed;
};

typedef struct BOOK BOOK;
int register_book(BOOK *book_list, int *nth);
int search_book(BOOK *book_list, int total_num_book);
int borrow_book(BOOK *book_list);
int return_book(BOOK *book_list);
int print_book_list(BOOK *book_list, int total_num_book);
int retrieve_book_info(BOOK **book_list, int *total_num_book);
char compare(char *str1, char *str2);

int main() {
  int user_choice;         /* 유저가 선택한 메뉴 */
  int num_total_book = 0; /* 현재 책의 수 */

  BOOK *book_list;
  int i;

  printf("도서관의 최대 보관 장서 수를 설정해주세요 : ");
  scanf("%d", &user_choice);

  book_list = (BOOK *)malloc(sizeof(BOOK) * user_choice);

  while (1) {
    printf("도서 관리 프로그램 \n");
    printf("메뉴를 선택하세요 \n");
    printf("1. 책을 새로 추가하기 \n");
    printf("2. 책을 검색하기 \n");
```

```
    printf("3. 책을 빌리기 \n");
    printf("4. 책을 반납하기 \n");
    printf("5. 프로그램 종료 \n");
    printf("6. 책들의 내용을 book_list.txt 에 출력 \n");
    printf("7. 책들의 내용을 book_list.txt 에서 불러옴 \n");
    printf("8. 책들의 목록을 출력 \n");

    printf("당신의 선택은 : ");
    scanf("%d", &user_choice);

    if (user_choice == 1) {
      /* 책을 새로 추가하는 함수 호출 */
      register_book(book_list, &num_total_book);
    } else if (user_choice == 2) {
      /* 책을 검색하는 함수 호출 */
      search_book(book_list, num_total_book);
    } else if (user_choice == 3) {
      /* 책을 빌리는 함수 호출 */
      borrow_book(book_list);
    } else if (user_choice == 4) {
      /* 책을 반납하는 함수 호출 */
      return_book(book_list);
    } else if (user_choice == 5) {
      /* 프로그램을 종료한다. */
      break;
    } else if (user_choice == 6) {
      /* book_list.txt 에 책들의 목록을 출력한다*/
      print_book_list(book_list, num_total_book);
    } else if (user_choice == 7) {
      /* book_list.txt 에서 책들의 목록을 가져온다*/
      retrieve_book_info(&book_list, &num_total_book);
    } else if (user_choice == 8) {
      /* 책들의 목록을 화면에 출력한다. */
      for (i = 0; i < num_total_book; i++) {
        printf("%s // %s // %s // ", book_list[i].book_name,
               book_list[i].auth_name, book_list[i].publ_name);
        if (book_list[i].borrowed == 0)
          printf("NO\n");
        else
          printf("YES\n");
      }
    }
  }

  free(book_list);
  return 0;
}
int print_book_list(BOOK *book_list, int total_num_book) {
  FILE *fp = fopen("book_list.txt", "w");
  int i;
```

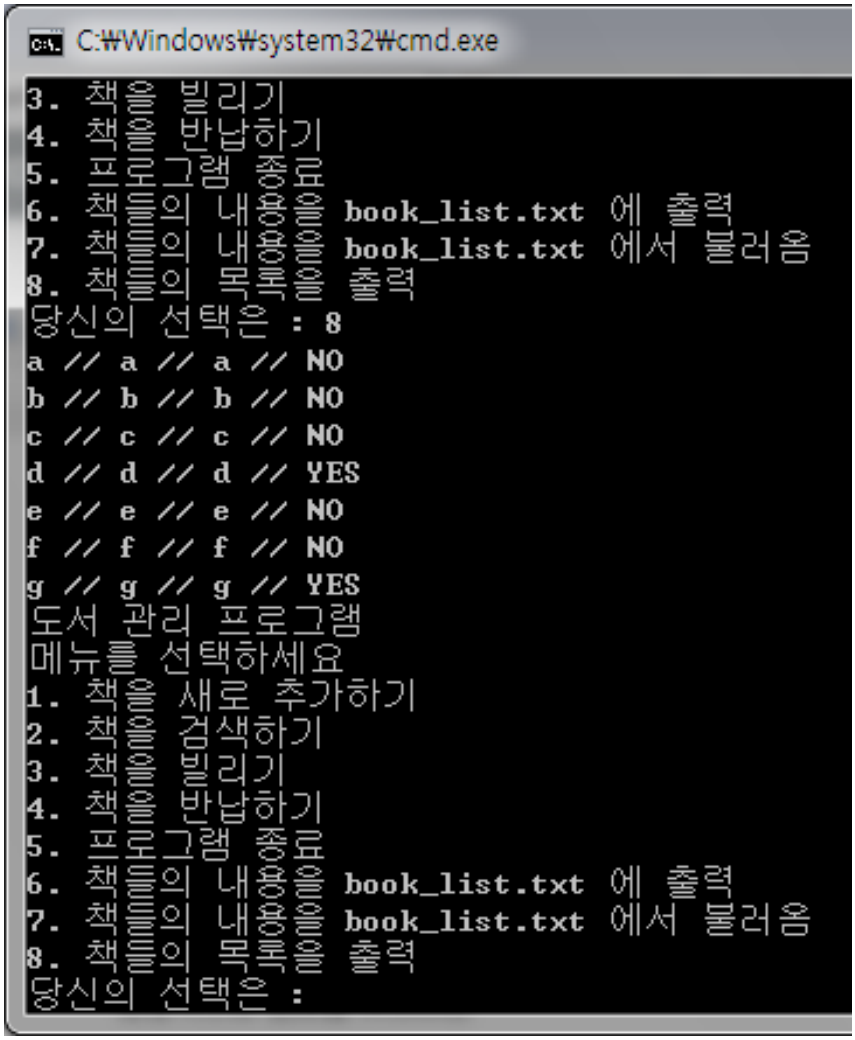
```
  if (fp == NULL) {
    printf("출력 오류 ! \n");
    return -1;
  }

  fprintf(fp, "%d\n", total_num_book);

  for (i = 0; i < total_num_book; i++) {
    fprintf(fp, "%s\n%s\n%s\n", book_list[i].book_name, book_list[i].auth_name,
            book_list[i].publ_name);
    if (book_list[i].borrowed == 0)
      fprintf(fp, "NO\n");
    else
      fprintf(fp, "YES\n");
  }

  printf("출력 완료! \n");
  fclose(fp);

  return 0;
}
char compare(char *str1, char *str2) {
  while (*str1) {
    if (*str1 != *str2) {
      return 0;
    }

    str1++;
    str2++;
  }

  if (*str2 == '\0') return 1;

  return 0;
}

/* 포인터인 book_list 의 값을 바꿔야 하므로 더블 포인터 형태 */
int retrieve_book_info(BOOK **book_list, int *total_num_book) {
  FILE *fp = fopen("book_list.txt", "r");
  int total_book;
  int i;
  char str[10];

  if (fp == NULL) {
    printf("지정한 파일을 찾을 수 없습니다! \n");
    return -1;
  }

  /* 찾았다면 전체 책의 개수를 읽어온다. */
  fscanf(fp, "%d", &total_book);
  (*total_num_book) = total_book;
```

```
  /* 기존의 book_list 데이터를 삭제 */
  free(*book_list);
  /* 그리고 다시 malloc 으로 재할당 한다. */
  (*book_list) = (BOOK *)malloc(sizeof(BOOK) * total_book);

  if (*book_list == NULL) {
    printf("\n ERROR \n");
    return -1;
  }
  for (i = 0; i < total_book; i++) {
    /* book_list[i]->book_name 이 아님에 유의!! */
    fscanf(fp, "%s", (*book_list)[i].book_name);
    fscanf(fp, "%s", (*book_list)[i].auth_name);
    fscanf(fp, "%s", (*book_list)[i].publ_name);
    fscanf(fp, "%s", str);

    if (compare(str, "YES")) {
      (*book_list)[i].borrowed = 1;
    } else if (compare(str, "NO")) {
      (*book_list)[i].borrowed = 0;
    }
  }

  fclose(fp);
  return 0;
}
/* 책을 추가하는 함수*/
int register_book(BOOK *book_list, int *nth) {
  printf("책의 이름 : ");
  scanf("%s", book_list[*nth].book_name);

  printf("책의 저자 : ");
  scanf("%s", book_list[*nth].auth_name);

  printf("책의 출판사 : ");
  scanf("%s", book_list[*nth].publ_name);

  book_list[*nth].borrowed = 0;
  (*nth)++;

  return 0;
}
/* 책을 검색하는 함수 */
int search_book(BOOK *book_list, int total_num_book) {
  int user_input; /* 사용자의 입력을 받는다. */
  int i;
  char user_search[30];

  printf("어느 것으로 검색 할 것인가요? \n");
  printf("1. 책 제목 검색 \n");
```

```
  printf("2. 지은이 검색 \n");
  printf("3. 출판사 검색 \n");
  scanf("%d", &user_input);

  printf("검색할 단어를 입력해주세요 : ");
  scanf("%s", user_search);

  printf("검색 결과 \n");

  if (user_input == 1) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 책 제목을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < total_num_book; i++) {
      if (compare(book_list[i].book_name, user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_list[i].book_name, book_list[i].auth_name,
               book_list[i].publ_name);
      }
    }

  } else if (user_input == 2) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 지은이 이름을
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < total_num_book; i++) {
      if (compare(book_list[i].auth_name, user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_list[i].book_name, book_list[i].auth_name,
               book_list[i].publ_name);
      }
    }

  } else if (user_input == 3) {
    /*

    i 가 0 부터 num_total_book 까지 가면서 각각의 출판사를
    사용자가 입력한 검색어와 비교하고 있다.

    */
    for (i = 0; i < total_num_book; i++) {
      if (compare(book_list[i].publ_name, user_search)) {
        printf("번호 : %d // 책 이름 : %s // 지은이 : %s // 출판사 : %s \n", i,
               book_list[i].book_name, book_list[i].auth_name,
               book_list[i].publ_name);
```

```
      }
    }
  }

  return 0;
}
int borrow_book(BOOK *book_list) {
  /* 사용자로 부터 책번호를 받을 변수*/
  int book_num;

  printf("빌릴 책의 번호를 말해주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &book_num);

  if (book_list[book_num].borrowed == 1) {
    printf("이미 대출된 책입니다! \n");
  } else {
    printf("책이 성공적으로 대출되었습니다. \n");
    book_list[book_num].borrowed = 1;
  }

  return 0;
}
int return_book(BOOK *book_list) {
  /* 반납할 책의 번호 */
  int num_book;

  printf("반납할 책의 번호를 써주세요 \n");
  printf("책 번호 : ");
  scanf("%d", &num_book);

  if (book_list[num_book].borrowed == 0) {
    printf("이미 반납되어 있는 상태입니다\n");
  } else {
    book_list[num_book].borrowed = 0;
    printf("성공적으로 반납되었습니다\n");
  }

  return 0;
}
```

성공적으로 컴파일 하였다면



와 같이 여러가지 기능들이 잘 작동함을 알 수 있습니다.

일단, 저는 지난번 강좌의 '생각해보기' 에서 일컫은대로 `print_book_list` 함수의 파일 출력 형태를 수정하였습니다.

```
int print_book_list(BOOK *book_list, int total_num_book) {
  FILE *fp = fopen("book_list.txt", "w");
  int i;

  if (fp == NULL) {
    printf("출력 오류 ! \n");
    return -1;
  }

  fprintf(fp, "%d\n", total_num_book);

  for (i = 0; i < total_num_book; i++) {
    fprintf(fp, "%s\n%s\n%s\n", book_list[i].book_name, book_list[i].auth_name,
            book_list[i].publ_name);
    if (book_list[i].borrowed == 0)
```

```
      fprintf(fp, "NO\n");
    else
      fprintf(fp, "YES\n");
  }

  printf("출력 완료! \n");
  fclose(fp);

  return 0;
}
```

일단 출력을 위해 fprintf 함수를 사용하고 있는데 문자열을 하나 출력할 때 마다 \n 을 넣어서 나중에 파일에서 입력받을 때 구분을 용이하게 하였습니다. print_book_list 함수는 상당히 간단하므로 설명은 이 정도에서 마치도록 하겠고요, 이제 가장 중요한 부분인 retrieve_book_list 함수에 대해 살펴봅시다.

```
/* 포인터인 book_list 의 값을 바꿔야 하므로 더블 포인터 형태 */
int retrieve_book_info(BOOK **book_list, int *total_num_book) {
  FILE *fp = fopen("book_list.txt", "r");
  int total_book;
  int i;
  char str[10];

  if (fp == NULL) {
    printf("지정한 파일을 찾을 수 없습니다! \n");
    return -1;
  }

  /* 찾았다면 전체 책의 개수를 읽어온다. */
  fscanf(fp, "%d", &total_book);
  (*total_num_book) = total_book;

  /* 기존의 book_list 데이터를 삭제 */
  free(*book_list);
  /* 그리고 다시 malloc 으로 재할당 한다. */
  (*book_list) = (BOOK *)malloc(sizeof(BOOK) * total_book);

  if (*book_list == NULL) {
    printf("\n ERROR \n");
    return -1;
  }
  for (i = 0; i < total_book; i++) {
    /* book_list[i]->book_name 이 아님에 유의!! */
    fscanf(fp, "%s", (*book_list)[i].book_name);
    fscanf(fp, "%s", (*book_list)[i].auth_name);
    fscanf(fp, "%s", (*book_list)[i].publ_name);
    fscanf(fp, "%s", str);

    if (compare(str, "YES")) {
```

```
      (*book_list)[i].borrowed = 1;
    } else if (compare(str, "NO")) {
      (*book_list)[i].borrowed = 0;
    }
  }

  fclose(fp);
  return 0;
}
```

이 함수를 처음에 만들때 저는 살짝 실수를 했었는데요, 그것은 바로 인자의 형태를 잘못 생각하였기 때문입니다. 일단 이 함수에서 하는 일들을 살펴보자면

1. 기존의 프로그램상에 저장되어 있던 book_list 정보를 없앤다. (간단히 메모리를 free 한다고 생각하시면 됩니다)

1. 파일에서 입력받은 책의 수 많큼 malloc 으로 book_list 에 할당한다.

1. 파일에 책들에 관한 데이터들을 입력받는다.

으로 보실 수 있습니다. 먼저 1 번 작업을 하는 부분을 살펴봅시다.

```
int retrieve_book_info(BOOK **book_list, int *total_num_book) {
  /*  ...(생략) ...*/

  /* 찾았다면 전체 책의 개수를 읽어온다. */
  fscanf(fp, "%d", &total_book);
  (*total_num_book) = total_book;

  /* 기존의 book_list 데이터를 삭제 */
  free(*book_list);
```

함수의 인자 부분이 굵은 글씨체로 되어 있는데 아마도 여러분들 중 많은 분들이 간과했을 부분입니다. 우리는 이 함수에서 malloc 을 통해 book_list 를 다시 할당 하면서 book_list 의 값을 바꿔야 하는데요,이전의 저의 함수 강좌를 잘 보신 분들은 알겠지만 다른 함수에서 또다른 함수 내부에서 정의된 변수의 값을 바꾸기 위해서는 그 변수의 포인터를 인자로 전달해야 한다고 했습니다. 즉, book_list 를 가리키는 포인터, 이 때 book_list 가 BOOK* 형이므로 우리는 인자를 BOOK** 형으로 잡아야 한다는 것을 알 수 있지요.

이제 인자로 book_list 를 받았다면 위와 같이 free 를 통해 메모리 공간을 반환 함으로써 이전에 저장되어 있는 책 목록 데이터를 없앨 수 있습니다.

```
/* 그리고 다시 malloc 으로 재할당 한다. */
(*book_list) = (BOOK *)malloc(sizeof(BOOK) * total_book);
```

```
if (*book_list == NULL) {
  printf("\n ERROR \n");
  return -1;
}
```

이제 두 번째 단계를 살펴보도록 합시다. 아까 위에서 free 함수를 통해 우리가 동적으로 할당하였던 Heap 공간에 있었던 데이터들은 날라가버렸습니다. 이제 다시 malloc 을 통해 책 목록에서 입력받는 total_book 의 크기 만큼 다시 할당 해야 겠지요? 따라서 위와 같이 malloc 함수를 통해 할당할 수 있게 됩니다.

여기서 중요한 부분은 위와 같이 *book_list == NULL 을 통해 malloc 으로 메모리 할당이 제대로 이루어졌는지 확인할 수 있습니다. 이는 마치 파일 입출력에서 fp == NULL 로 검사하는 것과 비슷한 이치 입니다.

```
for (i = 0; i < total_book; i++) {
  /* book_list[i]->book_name 이 아님에 유의!! */
  fscanf(fp, "%s", (*book_list)[i].book_name);
  fscanf(fp, "%s", (*book_list)[i].auth_name);
  fscanf(fp, "%s", (*book_list)[i].publ_name);
  fscanf(fp, "%s", str);

  if (compare(str, "YES")) {
    (*book_list)[i].borrowed = 1;
  } else if (compare(str, "NO")) {
    (*book_list)[i].borrowed = 0;
  }
}
```

마지막으로 for 문을 통해서 파일로 부터 입력 받는 부분을 살펴봅시다. 여기서 fscanf 를 통해 문자열을 파일로 부터 입력을 받고 있는데 재미있는 부분은 book_list 의 구조체 변수들을 참조하는 과정 입니다. 왜 저렇게 어렵게 표현을 했을까요?

일단 *book_list 를 통해서 원래 main 함수의 book_list 배열을 가리킬 수 있습니다. 그리고 (*book_list)[i] 를 통해 그 배열의 i 번째 원소를 가리킬 수 있지요. 이 때 그 원소는 구조체 변수 입니다. 따라서 우리는 (*book_list)[i].book_name 을 통해 main 함수의 특정 책의 책 이름에 대한 정보를 지칭할 수 있게 되지요. 이를 전개해서 표현 시켜 보면

```
(*(*book_list) + i)).book_name
```

이라 쓴 것과 동일한 표현이 됩니다. 그런데 문제는 이와 같이 포인터를 이용할 때 간혹 다음과 같이 쓰는 사람들이 있기 마련이죠.

```
book_list[i]->book_name
```

사실 저도 처음에 이렇게 썼다가 봉변을 맞았었는데 이렇게 사용하면 컴파일 시에 오류가 나지도 않고 디버깅 해도 도대체 뭐가 문제인지 알기 어렵습니다. 원래는 `(*book_list)[i].book_name` 이라고 머리 속에 생각은 하고 위와 같은 코드가 튀어 나왔지요. 그런데 이 둘은 완전히 다른 문장입니다. 무엇이 다른지는 말그대로 전개를 시켜보면 알 수 있습니다.

```
(*book_list)[i].book_name == (*(*book_list + i)).book_name;
book_list[i]->book_name == (*(*(book_list + i))).book_name;
```

무엇이 다른지 확 눈에 보이시나요? 위의 식에서는 `book_list` 에 먼저 `*` 이 붙었고 아래 식에서는 `book_list + i` 에 먼저 `*` 이 붙었습니다. 우리가 원하던 내용은 `book_list` 에 `*` 이 붙어서 `main` 함수의 실제 `book_list` 를 의미한 뒤 `+ i` 를 해서 원소를 참조하는 것이지 후자 처럼 존재하지도 않는 `book_list` 의 `i` 번째 원소를 참조하여 말도 안되는 식을 의도하는 것은 아닙니다.

아무튼 이렇게 해서 성공적으로 도서 관리 프로그램에 대한 파일 입출력을 수행 할 수 있게 됩니다.

이것으로 해서 파일 입출력에 관한 강좌는 끝내도록 하겠습니다. 기타 파일 입출력에 관한 함수들에 대해 설명하는 것은 무의미한 행동이므로 파일 입출력에 관련한 함수들을 더욱 알고 싶다면 C 언어 레퍼런스를 참조하시기 바랍니다 (특히 보셔야 할 부분은 `stdio.h` 부분이지요).

이제 여기 까지 도달하셨다면 여러분은 C 언어에 대한 기초적인 모든 부분을 학습하셨다고 해도 과언이 아닙니다. 여러분의 손에는 이제 훌륭한 조각칼이 들려있는 셈입니다. 이 조각칼로 여러분의 능력에 따라 멋진 조각품을 만들어 낼 수 있듯이 C 언어라는 훌륭한 도구를 통해서 최고의 프로그램을 만들 수 있게 될 것입니다.

저의 C 언어 강좌는 이제 마지막 강좌를 남겨두고 있습니다. 사실 여기서 강좌를 끝내도 무방하고, 여러분들은 여기까지 강좌를 읽으셨다면 축하 파티를 하셔도 됩니다! 마지막 강좌는여러분들이 실제로 프로그래밍을 하면서 여러가지 실수들을 하실 텐데 이를 대비하기 위한 오류 해결책들에 대해 이야기 해보려고 합니다. 여러 골치 아픈 오류들을 만났을 때 꽤 도움이 많이 될 듯 합니다. 여러분들이 실제로 프로그래밍을 하면서 피가 되고 살이 될 '코드 최적화' 에 대해서 이야기 하고자 합니다. 우리가 흔히 느끼기에 똑같은 한국말을 해도 어떤 사람은 말을 참 잘하지만 어떠한 사람은 말을 참 못한다고 생각 하는 경우가 있습니다. C 언어도 마찬가지 입니다. 저의 C 언어 강좌는 여기서 끝났지만 C 언어 자체를 안다고 해도 모두다 훌륭한 코드를 쓸 수 있는 것은 아닙니다. 따라서 다음 강좌에서는 여러분이 훌륭한 코드에 조금이니마 근접한 코드를 쓸 수 있도록 도와 줄 코드 최적화에 대해서 이야기 할 것입니다.

생각해보기

문제 1

책들의 목록을 `html` 형식에 맞게 출력하여 표로 깔끔하게 보여질 수 있게 해보세요. `html` 문법은 http://www.w3schools.com/html/default.asp에서 배우실 수 있습니다. (난이도 : 上)

22

C 코드 최적화

안녕하세요 여러분~ 이제 저의 마지막 강의(총 41 강)가 되겠네요. 그럼, 오늘도 강의를 시작해 볼까요?

우리의 컴퓨터는 무한정 빠르지 않습니다. 따라서 동일한 작업을 시키더라도 어떠한 방식으로 시키냐에 따라서 그 속도가 엄청나게 차이가 나게 됩니다. 우리는 언제나 코드를 만들 때 '과연 어떻게 해야지 이 작업을 가장 빠르게 할 수 있도록 코드를 만들 수 있을까?' 를 고민 해야 합니다. 이렇게 똑같은 일이라도 더 빠르게 수행할 수 있도록 코드를 짜는 행위를 **코드 최적화** 라고 부릅니다.

참고적으로 아래의 내용은 대부분 여기에서 가져왔으며, 특히 한국어로 번역된 자료는 여기에서 보실 수 있습니다. 저는 여러분들께 여기에서 가장 중요하다고 생각되는 몇 가지 부분만을 쉽게 이야기 하고자 합니다.

당부의 말

프로그램 최적화는 매우 어려운 작업입니다. 사실 아래에서 소개하는 것들은 매우 간단한 것들만 소개하는 것이고, **현대의 컴파일러의 경우 아마 여러분 보다 훨씬 더 최적화를 잘할 것입니다.** (그래서 그냥 놔두는 것이 더 빠른 경우가 많습니다!)

제가 권하고 싶은 말은 아래 최적화 방식을 맹신하라는 뜻은 아니고 아 이러한 방식으로도 생각해볼 수 있구나 정도가 매우 좋을 것 같습니다.

또한 최적화를 했다고 해도 실제로 성능이 향상될 수 있는지는 아무도 모릅니다. 프로그램 작동 속도에 영향을 줄 수 있는 것들이 무수히 많기 때문이죠. 따라서, 항상 실제 향상 속도를 언제나 테스트 해보는 것이 중요합니다. (이를 프로파일링(profiling) 이라 합니다)

산술 연산 관련

부동 소수점 (float, double) 은 되도록 사용하지 말자

예전에 10 강에서 부동 소수점 수의 구조에 대해 이야기 한 적이 있습니다. 그 때 강좌를 잘 보셨던 분은 알겠지만 부동 소수점 수는 그 구조가 매우매우 복잡합니다. 정수 자료형(`int`, `short`, ...) 의 경우 단순히 2 진수를 나타낸 것에 불과하지만 부동 소수점은 그 수의 특정한 규격이 정해진 것이기 때문에 상당히 복잡하지요.

따라서 부동 소수점 수를 가지고 하는 연산 자체도 매우 느릴 수 밖에 없습니다. 여러분들은 꼭부동 소수점 연산은 오직 **반드시 필요할 때 에만** 사용하시기 바랍니다. 여기서 반드시 필요할 때 라면 소수점 몇 째 자리 까지 정밀도를 요구할 때에나 매우 큰 수를 다룰 때 입니다. 만일 소수점 둘째 자리나 첫째 자리 정도의 정밀도를 요구한다면 단순히 그 수에 `x 10, x 100` 을 하셔서 정수 자료형으로 다루는 것이 오히려 좋습니다.

나눗셈을 피해라 (1)

아래는 초 를 증가시켜주는 함수 입니다.

```
int inc_second(int second) { return (++second) % 60; }
```

초의 범위는 0 부터 59 이므로 1 증가시킨 뒤에 만일 60 을 넘었을 때를 대비하여 위와 같이 60 으로 나눈 나머지를 구해야 되겠지요. 그런데 여기서 문제는 나눗셈은 매우매우 느린 연산이라는 것입니다.

다른 덧셈 뺄셈에 비해 몇 배 가까이 느리기 때문에 엄청난 시간 손해가 있겠지요. 우리가 만약 `second` 가 60 보다 커질 일이 없다는 것을 알고 있다면 굳이 60 으로 나눌 필요 없이 `if` 문으로 60 일 때만 0 을 리턴해주면 되는 것입니다. 왜냐하면 `if` 문은 나눗셈 보다는 훨씬 빠르게 처리가 되기 때문이지요.

```
int inc_second(int second) {
  ++second;
  if (second >= 60) return 0;
  return second;
}
```

따라서 위와 같이 하면 훨씬 시간을 아낄 수 있습니다.

한 가지 짚고 넘어갈 점은 위 코드에서 분기문 (`if`) 를 도입하였다는 점입니다. 때로는 분기문이 프로그램 속도를 저하시킬 수 있습니다.

왜냐하면 CPU 의 경우 명령어 실행 속도를 향상시키기 위해 파이프라이닝 이라는 작업을 수행합니다. 쉽게 말하자면, 다음에 실행될 명령어를 이전 명령어 실행이 채 끝나기 전에 미리 실행시키는 것과 비슷하다고 보면 됩니다.

문제는 분기문이 있을 경우 다음에 실행할 명령어가 무엇인지 모른다는 점입니다. 위 경우 `second >= 60` 이면 `return 0;` 명령을 수행해야 되고 아니면 `return second` 명령을 수행해야 한다는 점이지요.

그렇다면 CPU 가 `second >= 60` 이 끝날 때 까지 기다릴까요? 아닙니다. 이전에 추세를 보아서 대충 참일지 거짓일지 **예측** 한 다음 다음에 올 명령어를 실행하게 됩니다. 이렇게 분기문을 예측하는 것을 **분기 예측(branch prediction)** 이라 합니다.

예측이 맞았다면 기분좋게 쭉쭉 진행할 수 있었지만, 예측이 틀렸더라면 여태까지 작업한 것을 모두 버리고 원래 수행했어야 할 명령어를 다시 실행해야 합니다. Intel Skylake CPU 의 경우 해당 페널티가 20 cycle 정도 됩니다. 참고로 정수 나눗셈 연산(DIV) 의 경우 10 cycle 정도 필요하고, 덧셈의 경우 1 cycle 에 끝나게 됩니다. (즉 나눗셈이 덧셈 보다 10 배 느립니다.)[1)]

따라서 만약에 분기 예측 정확도를 50% 이상으로 할 수 만 있다면 위와 같이 코드를 바꿨을 때 효율적으로 최적화를 했다고 볼 수 있습니다. 다행이도 CPU 의 분기 예측기는 꽤나 똑똑해서, **이전의 분기 결과에 추세** 를 바탕으로 예측을 하게 됩니다. 우리의 `inc_second` 는 대부분의 경우 `second >= 60` 이 참일 일이 없기 때문에 (확률상 `1/60` 이다), 분기 예측 확률이 꽤 높을 것입니다.

물론 실제 프로그램에서 `inc_second` 가 어떻게 사용되는지는 아무도 모릅니다. 따라서 반드시 테스트를 통해 실제 향상이 있는지 확인해보는 것이 좋습니다.

나눗셈을 피해라 (2)

대부분의 현대 컴파일러들은 이 작업을 알아서 최적화 해줍니다.

앞에서도 말했듯이 나눗셈은 시간이 매우매우 오래 걸리는 작업이라고 했습니다. 그런데 놀랍게도 2 의 멱수들 (2,4,8,16,32 ...) 로 나눌 때에는 굳이 나눗셈을 사용하지 않고도 매우 간단하게 처리할 수 있는 방법이 있습니다. 바로 '쉬프트' 연산을 사용하는 것입니다. 쉬프트는 컴퓨터 연산 중에서도 가장 빠른 연산이므로 이를 잘만 활용한다면 시간을 엄청나게 절약할 수 있습니다.

2 의 멱수들을 이진수로 표현해 보면 1, 10, 100, 1000 등이 될 것입니다. 그럼 감이 오시나요? 우리가 만일 10 진수로 생각할 때 7865 를 100 으로 나누면 몫이 얼마가 될까요? 우리는 별로 고민하지 않고도 78 이라고 말할 수 있을 것입니다. 왜냐하면 단순히 끝의 두 자리를 버려버리면 되기 때문이지요. 이진수도 마찬가지 입니다. 11101010 을 1000 으로 나눈 몫은 얼마일까요? 이는 단순히 마지막 세자리를 버리면 되므로 11101 이 되겠지요.

1) Cycle 이란 쉽게 생각하면 CPU 에서 한 가지 작업을 수행하는데 걸리는 시간 단위라 보시면 됩니다. 만약에 내 CPU 가 4Ghz 라면 1초에 40 억 cycle 어치의 연산을 수행할 수 있습니다

이 아이디어를 이용하면 1000 (이진수) 으로 나눌 때 에는 수를 오른쪽으로 3 칸 쉬프트 해버리면 됩니다. 즉, 오른쪽으로 3 칸 밀어버리는 것이지요 (쉬프트가 기억이 나지 않으면 4 강 계산하리를 보시기 바랍니다) 아래 예제는 32 로 나누는 것입니다. 32 는 2 의 5 승이므로 오른쪽으로 5 칸 쉬프트 해버리면 됩니다.

```c
#include <stdio.h>
int main() {
  int i;
  printf("정수를 입력하세요 : ");
  scanf("%d", &i);

  printf("%d 를 32 로 나누면 : %d \n", i, i / 32);
  printf("%d 를 5 칸 쉬프트 하면 : %d \n", i, i >> 5);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
정수를 입력하세요 : 120
120 를 32 로 나누면 : 3
120 를 5 칸 쉬프트 하면 : 3
```

두 결과가 일치함을 보실 수 있습니다.

비트 연산 활용하기 (1)

비트 연산(OR, AND, XOR 등등) 은 컴퓨터에서 가장 빠르게 실행되는 연산들 입니다. 이러한 연산들을 잘 활용하면 좋겠지요. 일단 비트연산은 다음과 같이 여러가지 정보를 하나의 변수에 포함하는데 자주 사용됩니다. 예를 들어서 우리가 하나의 사람에 대한 여러가지 상태에 관한 정보를 나타내는 변수를 만든다고 합시다. 구조체를 배운 여러분으로써는 아래와 같이 만들 것입니다.

```c
struct HUMAN {
  int is_Alive;
  int is_Walking;
  int is_Running;
  int is_Jumping;
  int is_Sleeping;
  int is_Eating;
};
```

이는 상당한 메모리 낭비가 되겠지요. 6 가지 정보를 나타내는데 192 개의 비트나 소모하였기 때문이지요. 물론 이를 char 로 바꾸면 되지 않나 라고 물어볼 수 있지만 결국은 같은 얘기 입니다.

굳이 하나의 정보를 한 개의 비트에 대응시켜서 사용할 수 도 있는데 이를 각각의 변수에 모두 대응시켜서 사용한 것이 문제이지요. 하지만 비트 연산을 잘 이용하면 이를 해결할 수 있습니다. 아래의 예제를 보세요

```c
#include <stdio.h>
#define ALIVE 0x1       // 2 진수로 1
#define WALKING 0x2     // 2 진수로 10
#define RUNNING 0x4     // 2 진수로 100
#define JUMPING 0x8     // 2 진수로 1000
#define SLEEPING 0x10   // 2 진수로 10000
#define EATING 0x20     // 2 진수로 100000
int main() {
  int my_status = ALIVE | WALKING | EATING;

  if (my_status & ALIVE) {
    printf("I am ALIVE!! \n");
  }
  if (my_status & WALKING) {
    printf("I am WALKING!! \n");
  }
  if (my_status & RUNNING) {
    printf("I am RUNNING!! \n");
  }
  if (my_status & JUMPING) {
    printf("I am JUMPING!! \n");
  }
  if (my_status & SLEEPING) {
    printf("I am SLEEPING!! \n");
  }
  if (my_status & EATING) {
    printf("I am EATING!! \n");
  }
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
I am ALIVE!!
I am WALKING!!
I am EATING!!
```

와 같이 단순히 하나의 int 변수에 위 모든 데이터를 나타낼 수 있었습니다. 그 이유는 아래와 같이

```
#define ALIVE 0x1     // 2 진수로 1
#define WALKING 0x2   // 2 진수로 10
#define RUNNING 0x4   // 2 진수로 100
#define JUMPING 0x8   // 2 진수로 1000
#define SLEEPING 0xC  // 2 진수로 10000
#define EATING 0x10   // 2 진수로 100000
```

define 을 이용해 여러개의 변수에 값을 대응시켰는데 한가지 특징은 각 데이터에는 오직 한 개의 비트만 1 이고 나머지는 모두 0 인 것입니다. 예를 들면 JUMPING 을 보면 16 진수 8 을 대응시켰는데, 이를 2 진수로 보면 끝에서 4 번째 자리만 1 이고 나머지 모든 자리는 0 인 수가 됩니다. 따라서 이와 같은 방식으로 수를 대응시키고

```
int my_status = ALIVE | WALKING | EATING;
```

와 같이 my_status 에 OR 연산을 시켜주게 되면 각 데이터들이 나타내는 자리만 1 이 되고 나머지 모든 자리는 0 이 됩니다. 따라서 my_status 에는 0...0100011 이 되겠지요. 이제 이를 이용하여 if 문에서도 쉽게 사용할 수 있는데 단순히 유무를 파악하고자 하는 데이터와 AND 연산을 시키면 됩니다.

```
if (my_status & WALKING) {
  printf("I am WALKING!! \n");
}
```

예를 들면 위와 같이 내가 현재 WALKING 중인지 아닌지 파악하기 위해 WALKING 과 AND 연산을 시켜보면 만일 내가 WALKING 중이였다면 AND 연산시 '나머지 부분은 모두 0 이고, WALKING 에 해당하는 자리수만 1 이 될 것' 이여서 if 문에서 참으로 판단되고 (if 문은 0 이 아닌 모든 값을 참으로 생각한다) , 내가 WALKING 중이 아니였다면 '나머지 부분은 모두 0 이고 WALKING 에 해당하는 자리수 조차 0 이 될 것' 이므로 0 이 되어서 if 문에서 거짓으로 판단됩니다.

참고로 비트 연산에 관련하여 아래의 내용을 기억하시면 편합니다.

1. 어떠한 정수의 특정 자리를 1 로 만들고 싶다면 그 자리만 1 이고 나머지는 0 인수와 OR 하면 됩니다.

1. 어떠한 정수의 특정 자리가 1 인지 검사하고 싶다면 그 자리만 1 이고 나머지는 0 인 수와 AND 하면 됩니다.

비트 연산 활용하기 (2)

비트 연산을 가장 많이 활용하는 예로 또한 홀수/짝수 판별이 있습니다. 여태까지 여러분들은 아마 홀수 짝수 판별을

```
if (i % 2 == 1)  // 이 수가 홀수인가
{
  printf("%d 는 홀수 입니다 \n", i);
} else {
  printf("%d 는 짝수 입니다 \n", i);
}
```

와 같이 만드셨을 것입니다. 그런데 제가 앞에서 계속 강조해 왔던 것이지만, 나눗셈 연산은 매우 느립니다! 하지만 놀랍게도 단순한 AND 연산 한번으로 이를 해결할 수 있습니다.

```
if (i & 1)  // 이 수가 홀수인가
{
  printf("%d 는 홀수 입니다 \n", i);
} else {
  printf("%d 는 짝수 입니다 \n", i);
}
```

만일 어떤 정수가 홀수라면, 2 진수로 나타냈을 때 맨 마지막 자리가 1 이여야 합니다.

이를 이용해서 단순히 어떤 정수의 맨 마지막 비트가 1 인지만 확인하면 되지요? 근데 위에서 강조했듯 것을 보면 맨 마지막 비트가 1 인지 확인하려면 맨 마지막 비트만 1 인 수(즉 1) 과 AND 하면 됩니다. 아래 소스로 컴파일해서 실행해보면 잘 됨을 알 수 있습니다.

```
#include <stdio.h>
int main() {
  int i;
  scanf("%d", &i);

  if (i & 1)  // 이 수가 홀수인가
  {
    printf("%d 는 홀수 입니다 \n", i);
  } else {
    printf("%d 는 짝수 입니다 \n", i);
  }
  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
33 는 홀수 입니다
```

루프(loop) 관련

알고 있는 일반적인 계산 결과를 이용하라

대표적으로 이야기 하자면 1 부터 n 까지 더하는 함수를 만들 때 입니다. 일반적으로 이러한 작업을 하는 코드를 짤 때에는

```
for (i = 1; i <= n; i++) {
  sum += i;
}
```

위와 같이 for 문으로 구현하는 경우가 대부분 입니다. 하지만 여러분이 초등학생 가우스 정도의 머리를 가졌더라면 위와 같이 일일히 더하는 것 말고도

```
sum = (n + 1) * n / 2;
```

로 간단히 나타낼 수 있겠지요. 이렇게 하게 될 경우 많은 계산 시간을 절약하게 됩니다.

끝낼 수 있을 때 끝내라

아래 코드는 특정한 문자열에 'a' 라는 문자가 포함되어 있는지 검사하는 코드 입니다.

```
while (*pstr) {
  if (*pstr != 'a') {
    does_string_has_a = 1;
  }

  pstr++;
}
```

위 코드에서 does_string_has_a 가 한 번 1 이 되었다면 뒤에서 바뀔 일이 없으므로 굳이 루프를 끝까지 실행하는 것은 무의미한 짓입니다. 이 때 이런 곳에 break 문을 넣어서 빠져 나갈 수 있게 한다면 불필요한 실행을 줄일 수 있습니다.

```
while (*pstr) {
  if (*pstr != 'a') {
    does_string_has_a = 1;
    break;
  }

  pstr++;
}
```

위 코드 처럼 말이지요.

한 번 돌 때 많이 해라.

하나의 루프에서 동일한 일을 2 번 하는 것과, 하나의 루프에서 동일한 일을 한 번 하고 루프를 두번 돈다면 전자의 경우가 훨씬 효율적이라 말할 수 있습니다. 왜냐하면 루프를 한 번 돌 때 여러가지 조건들이 맞는지 비교하는 부분에서 시간이 약간 소모되기 때문이지요. 따라서 되도록이면 루프 한 번에 안에서 많은 일을 해버리는 것이 중요합니다.

아래 코드는 정수 n 에서 값이 1 인 비트가 몇 개나 존재하는지 세는 프로그램 입니다.

```
while (n != 0) {
  if (n & 1) {
    one_bit++;
  }
  n >>= 1;
}
```

위 코드에서는 맨 끝 한개의 비트를 검사하고 오른쪽으로 쉬프트 해서 또 다시 맨 끝 비트를 검사하는 식으로 해서 결과적으로 모든 비트를 검사하여 값이 1 인 것의 개수를 셉니다. 하지만 우리는 C 언어에서 모든 정수 자료형의 크기가 8 비트의 배수 임을 알고 있습니다.

예를 들면 char 은 1 바이트로 8 비트, int 는 4 바이트로 32 비트 이지요. 따라서 굳이 1 개 비트씩 검사할 필요 없이 8 비트를 한꺼번에 묶어서 검사해도 상관이 없다는 말입니다. 이 때 8 비트를 한꺼번에 비교하면 너무 난잡하므로 4 비트씩 비교하는 것으로 하지요.

```
while (n != 0) {
  if (n & 1) {
    one_bit++;
  }
  if (n & 2) {
    one_bit++;
  }
  if (n & 4) {
    one_bit++;
  }
  if (n & 8) {
    one_bit++;
  }
  n >>= 4;
}
```

와 같이 하면 됩니다. 사실 C 언어에서 if 문이나 for 문 다음에 한 줄만이 올 경우 중괄호를 생략해도 되는데,

```
while (n != 0) {
  if (n & 1) one_bit++;
  if (n & 2) one_bit++;
  if (n & 4) one_bit++;
  if (n & 8) one_bit++;
  n >>= 4;
}
```

로 쓰셔도 됩니다. 아무튼 위와 같이 할 경우 루프 도는 회수를 줄일 수 있게 되므로 어느 정도의 시간 절약 효과를 보게 됩니다.

루프에서는 되도록 0 과 비교하여라

```
for (i = 0; i < 10; i++) {
  printf("a");
}

for (i = 9; i != 0; i--) {
  printf("a");
}
```

위 두 개의 for 문 중에서 무엇이 더 빠르게 실행될까? 실제로는 그리 큰 차이는 없을 테지만 엄밀히 따지고 보면 아래의 루프가 더 빠르게 돌아갑니다. 왜냐하면 위 루프의 경우 i 가 10 보다 작은지 비교하고 있고, 아래 루프에서는 i 가 0 과 다른지 비교하고 있는데 일반적으로 0 과 비교하는 명령어는 CPU 에서 따로 만들어져 있기 때문에 더 빠르게 작동될 수 있습니다.

되도록 루프를 적게 써라

루프문을 굳이 쓰지 않고 쓸 수 있는 문장들은 되도록 직접 쓰는 것이 좋습니다. 예를 들어

```
int i;
for (i = 1; i <= 3; i++) {
  func(i);
}
```

보다는

```
func(1);
func(2);
func(3);
```

와 같이 루프를 풀어버리는 것이 더 좋을 때가 있습니다. 물론 루프를 쓰면 무엇을 하는지 한눈에 알 수 있지만 `for` 문 자체에서 여러가지 비교를 수행하는데 시간이 들기 때문에 위와 같이 간단히 루프를 쓰지 않고도 나타낼수 있다면 그 방법을 선택하시기 바랍니다.

if 및 switch 문 관련

if 문을 2 의 배수로 쪼개기

예를 들면 아래와 같은 비교 명령들이 있다고 합시다.

```
if (i == 1) {
} else if (i == 2) {
} else if (i == 3) {
} else if (i == 4) {
} else if (i == 5) {
} else if (i == 6) {
} else if (i == 7) {
} else if (i == 8) {
}
```

(물론 위와 같은 명령들은 `switch` 문을 이용하는 것이 훨씬 바람직합니다)위 경우 `if` 문에서는 최악의 경우 최대 8 번의 비교작업을 해야 하는 상황이 발생합니다. 이는 엄청난 낭비가 아닐 수 없죠.

```
if (i <= 4) {
  if (i <= 2) {
    if (i == 1) {
      /* i is 1 */
    } else {
      /* i must be 2 */
    }
  } else {
    if (i == 3) {
      /* i is 3 */
    } else {
      /* i must be 4 */
    }
  }
} else {
  if (i <= 6) {
    if (i == 5) {
      /* i is 5 */
    } else {
      /* i must be 6 */
    }
```

```
  } else {
    if (i == 7) {
      /* i is 7 */
    } else {
      /* i must be 8 */
    }
  }
}
```

하지만 `if` 문을 위와 같이 구성하게 된다면 어떨까요? 이와 같이 `if` 문을 쪼개는 것을 `Binary Breakdown` 이라고 하는데 이진의 형태로 쪼갠 것이지요. 이럴 경우 `i` 가 1 에서 8 까지 어떠한 값을 가지더라도 3 번만의 비교로 값을 알아낼 수 있습니다. 참고로 이전의 `if` 문의 형태로는 평균적으로 4 번의 비교가 필요했지요.

순차적 비교에서는 switch 문을 사용해라

대부분의 현대 컴파일러들은 이 작업을 알아서 최적화 해줍니다.

사실 위의 `if` 문 예제에서는, 즉 위와 같이 순차적인 정수 값을 비교하는 경우에는 `switch` 문을 사용하는 것이 매우 요긴합니다. 왜냐하면 `switch` 문에서는 단 한번의 비교만으로 우리가 실행될 코드가 있는 곳으로 점프하기 때문이지요. `switch` 문의 원리는 9강에서 보시기 바랍니다.

즉 아래와 같은 코드가 훨씬 더 효율적입니다.

```
switch (i) {
  case 1:
    break;
  case 2:
    break;
  case 3:
    break;
  case 4:
    break;
  case 5:
    break;
  case 6:
    break;
  case 7:
    break;
  case 8:
    break;
}
```

룩업 테이블(look up table, LUT)을 사용할 수 있으면 사용해라

룩업 테이블이란, 원론적으로 설명하면 특정 데이터에서 다른 데이터로 변환할 때 사용되는 테이블이라 할 수 있습니다. 말만 들으면 조금 어려운데요, 사실 컴퓨터에서 매우 자주 사용되고 있습니다.

예를 들어 컴퓨터에서 3D 처리를 할 때 많은 수의 sine 이나 cosine 연산들이 들어가게 됩니다. 이 때 sin 값 계산은 꽤 오랜 시간 걸리는 계산인데 sin 1 값이 필요할 때 마다 계산을 하게 된다면 아주 시간 낭비가 심하겠지요. 이를 막기 위해 프로그램 실행 초기에 sin 1 부터 sin 90 까지 미리 다 계산해 둔 뒤 표로 만들어 버리면 나중에 sin 1 값이 필요하면 단순히 표에서 1 번째 값을 찾으면 되니까 아주 편하겠지요.

이렇게 만들어 놓은 테이블을 룩업 테이블이라고 부릅니다. 즉, 필요한 데이터를 쉽게 찾을 수 있도록 만들어 놓은 표라고 보시면 됩니다. 예를 들면 아래와 같은 경우 사용할 수 있습니다.

```
char* Condition_String1(int condition) {
  switch (condition) {
    case 0:
      return "EQ";
    case 1:
      return "NE";
    case 2:
      return "CS";
    case 3:
      return "CC";
    case 4:
      return "MI";
    case 5:
      return "PL";
    case 6:
      return "VS";
    case 7:
      return "VC";
    case 8:
      return "HI";
    case 9:
      return "LS";
    case 10:
      return "GE";
    case 11:
      return "LT";
    case 12:
      return "GT";
    case 13:
      return "LE";
    case 14:
      return "";
    default:
      return 0;
  }
```

```
}
```

위 코드의 경우 꽤 괜찮지만 아래 처럼 훨씬 간단하게 만들 수 있습니다.

```
char* Condition_String2(int condition) {
  if ((unsigned)condition >= 15) {
    return 0;
  }
  char* table[] = {"EQ", "NE", "CS", "CC", "MI", "PL", "VS",
                   "VC", "HI", "LS", "GE", "LT", "GT", "LE"};
  return table[condition];
}
```

이 때 위와 같이 룩업 테이블을 이용하면 좋은 점이 코드의 길이가 훨씬 짧아진다는 점이고 실제 프로그램의 크기도 줄어든다는 점에 있습니다.

함수 관련

함수를 호출할 때에는 시간이 걸린다.

```
#include <stdio.h> void print_a();

int main() {
  int i;
  for (i = 0; i < 10; i++) {
    print_a();
  }
  return 0;
}
void print_a() { printf("a"); }
```

위 코드와 아래 코드를 보면 무엇이 더 빠르게 작동할까요?

```
#include <stdio.h>
void print_a();
int main() {
  print_a();
  return 0;
}
void print_a() {
  int i;
  for (i = 0; i < 10; i++) {
    printf("a");
```

```
  }
}
```

그 답은 바로 아래 코드 입니다. 왜냐하면 함수를 호출하는 데에도 꽤 많은 시간이 걸리기 때문이지요. 함수를 호출하기 위해서는 여러가지 작업이 필요한데 이 부분에 대한 설명은 생략하고 아무튼 위와 같이 동일한 작업을 위해 함수를 반복적으로 호출하기 보단 차라리 그 함수 내에서 반복적인 작업을 처리하는 것이 훨씬 더 효율적입니다.

인라인(inline) 함수를 활용하자

```c
#include <stdio.h>
int max(int a, int b) {
  if (a > b) return a;
  return b;
}
__inline int imax(int a, int b) {
  if (a > b) return a;

  return b;
}
int main() {
  printf("4 와 5 중 큰 것은?", max(4, 5));
  printf("4 와 5 중 큰 것은?", imax(4, 5));
  return 0;
}
```

위 두 개의 `printf` 문 중에서 더 빠르게 실행되는 문장은 어떤 것일까요? 바로 아래의 `inline` 함수를 이용한 것입니다. 위와 같이 `max` 와 같은 단순한 작업을 함수로 만들 때 에는 인라인 함수를 사용하는 것이 훨씬 더 효율적입니다.

이미 잘 알고 계시겠지만 인라인 함수는 함수가 아닙니다. (자세한 설명은 21 강 – 매크로 함수, 인라인 함수참조). 반면에 `max` 함수는 실제로 함수의 호출 과정 부터 해서 여러가지 작업이 필요한데, 정작 내부에서 수행하는 작업은 매우 단순하여 오히려 함수 내부에서 하는 작업 시간 보다 호출하는데 걸리는 시간이 더 큰 배보다 배꼽이 더 큰 격이 됩니다. 따라서 위와 같이 단순한 작업을 함수로 만들 경우 인라인 함수를 이용하는 것이 더 좋습니다.

인자를 전달할 때에는 포인터를 이용해라

```c
struct big {
  int arr[1000];
  char str[1000];
};
```

위와 같은 매우 거대한 구조체가 있다고 합시다. 만일 이 구조체 변수의 arr[3] 값을 얻어 오는 함수를 만들고 싶다면 어떻게 해야 할까요? 물론 아래와 같이 프로그램을 짜는 사람도 있을 것입니다.

```
void modify(struct big arg) { /* 무언가를 한다 */ }
```

하지만 이 함수를 호출하게 될 경우 modify 함수의 arg 인자로 구조체 변수의 모든 데이터가 복사가 되야 하는데 이는 엄청난 시간이 걸리게 됩니다. 말그대로 5000 바이트나 되는 데이터의 복사를 수행해야 할 뿐더러 modify 변수의 메모리 공간을 위한 할당도 따로 필요하기 때문이지요. 그렇다면 아래의 코드는 어떨까요?

```
void modify(struct big *arg) { /* 무언가를 한다 */ }
```

위 함수는 구조체 변수의 주소값을 얻어옵니다. 이는 단순히 4 바이트의 주소값 복사만이 일어날 뿐 이전의 예와 같은 무지막지한 복사는 일어나지 않습니다. 뿐만 아니라 동일하게 인자로 전달된 구조체 변수의 데이터들도 손쉽게 읽어들일 수 있게 됩니다. 단순히 arg->arr[3] 과 같은 방식으로 말이지요. 여러분들은 언제나 이 점을 명심하시고 되도록 인자를 전달할 때에는 포인터를 자주 활용하시기 바랍니다.

그럼 이것으로 마지막 강좌를 끝내도록 하겠습니다~ 혹시 1 강 부터 시작해서 여기까지 도달하신 분이라면 https://forum.modoocode.com에 가셔서 꼭 글을 남겨주시기 바랍니다.

생각해보기

문제 1

다음의 글들을 읽어보세요

- http://decoder.tistory.com/529
- http://www.azillionmonkeys.com/qed/optimize.html

마침내 끝났습니다.

마침내 길고 길었던 항해가 끝났습니다. 아마도 이 글을 읽는 분들 중 저의 첫번째 강의부터 열심이 보셨던 분들이 여럿 계셨으면 하는 바램이 간절합니다. 제가 첫번째 강좌를 야심차게 시작했을 때가 2009 년 4월 16일 입니다.(사실 1 월 경에 시범적으로 C 언어 강좌 두 편을 올린 적이 있었는데 너무 어렵게 쓰는 바람에 반응이 없어서 다시 쓴 것입니다 ㅎ)

그리고 제가 마지막 강좌를 업로드 한 날짜가 2011 년 1 월 18일 이지요. 무려 2 년 간의 엄청난 격차가 있습니다. 평균적으로 따지면 1 달에 강좌 한 편도 올린 것이 아닌 셈인데, 제 블로그를 꾸준히 방문해 주신 여러분 덕분에 이렇게 강좌가 완성될 수 있었습니다.

감사의 말

아래 분들은 제 블로그에 (특히 더 많이) 기여를 해주신 분들입니다. 아래 여러분들께 저의 특별한 감사의 말을 전하고 싶습니다.

- 코이치 (39)
- 곰돌 (22)
- song (14)
- 감사합니다 (14)
- 희망 (11)
- 스프 (10)
- 프로그래머가 되고싶은 1인. (8)
- 괴도 (7)
- ore (7)
- 두루뭉술 (7)
- 질무이있으요. (6)
- 모르겠어요 (6)
- 배움의장터 (6)
- winape (6)
- 질문 (6)
- eager (5)
- 행인 (5)
- 궁금 (5)
- sweetick (5)
- Stephanos (5)
- 울림 (5)
- 공부중 (5)

그 외에 제 블로그를 방문해 주신 여러분!

감사합니다:)

씹어먹는 C 언어 칭찬

인터넷을 보다보면 수 많은 C 언어 강좌들이 있습니다. 하지만 안타까운 사실은 제가 본 대다수의 C 언어 강좌들은 대개 중간에 끝나는 경우가 엄청 많았습니다. 그 분들의 문제가 과연 무엇이였을까요? 개인적으로 생각했을 때 두 가지 문제가 있었다고 생각합니다. 먼저 강좌를 쓰는 일은 엄청난 끈기가 필요한 작업 입니다. 왜냐하면 자신이 완벽히 알고 있지 않는 한 남한테 가르쳐 주는 일은 매우 어려운 일이기 때문이지요. 예를 들면 포인터 하나를 가르치려고 해도, '포인터가 왜 필요한지' 부터 시작해서 '포인터의 타입은 왜 있는 것인지', 등등 모든 내용을 꿰뚫고 있어야 합니다. 그 만큼 가르치는 것은 엄청난 일이지요. 더군다나 얼굴 보고 맞대고 설명 하는 것도 힘든데 인터넷 상에서 글 만드로 이야기 할려니 얼마나 힘들겠습니까.

이를 이겨내더라도 문제가 한 가지 더 있습니다. 바로 읽는이와의 '소통' 이 불가능 하다라는 점입니다. 아무리 가르쳐 주는 사람이 훌륭하다고 해도 읽는 사람의 모든 부분을 이해시켜 줄 수 없는 것이 아닙니다. 한 마디로 한개의 강좌로 독자의 가려운 곳을 정확하게 긁어줄 수 없는 셈이지요. 이를 위해서는 확실한 '애프터 서비스' 가 필요한데 이것이 바로 독자와 '댓글' 을 통해 소통하는 것이였습니다. 제 강좌에서 가장 강조 했던 부분이 바로 이 부분이였습니다. 강좌에서 궁금한 점이나 이해가 안가는 점이 있다면 무조건 '댓글을 남겨라' 이지요. 이렇게 댓글을 남겨서 이를 해결해 주는 방법을 통해서 저는 제 글을 읽는 여러분 모두와 확실하게 소통을 할 수 있었다고 생각합니다. 바로 이 것이 1 달에 강좌를 평균적으로 1 편 씩만을 써도 끝까지 완결할 수 있는 방법이지요 :)

아무튼 이제 여러분은 'C 언어' 라는 하나의 산을 정ㅋ벅ㅋ 하셨습니다.

그럼 여러분은 이제 무얼 해야 될까요?

일단 여러가지를 하실 수 있습니다. 만일 C 언어를 다른 곳에서 이미 배우시고 제 강좌를 '복습' 차원에서 읽으신 분들이라면 제 강좌를 다시 볼 필요가 없겠지요. 하지만 C 언어를 제 강좌를 통해 처음 접했던 분들이라면 복습이 최고라고 말씀드리고 싶습니다. 물론 여기서 복습이란 말은 다시 꼼꼼히 정독 하라는 것이 아니라 대충 필요한 강좌만 다시 보는게 좋을 것이라는 거지요.

아래는 저의 전체 강좌 리스트 입니다.

1. 2009/04/16 씹어먹는 C 언어 - <1. C 언어가 뭐야?>
2. 2009/04/17 씹어먹는 C 언어 - <2 - 1. C 언어 본격 맛보기>
3. 2009/09/24 씹어먹는 C 언어 - <2 - 2. 주석(Comment)에 대한 이해>
4. 2009/10/12 씹어먹는 C 언어 - <2 - 3. 수를 표현하는 방법(기수법)>
5. 2009/04/22 씹어먹는 C 언어 - <3. 변수가 뭐지? >
6. 2009/04/24 씹어먹는 C 언어 - <4. 계산하리 >
7. 2009/04/27 씹어먹는 C 언어 - <5. 문자 입력 받기>

8. 2009/04/28 씹어먹는 C 언어 - <6. 만약에...(if 문)>
9. 2009/08/06 씹어먹는 C 언어 - <7. 뱅글 뱅글 (for, while) >
10. 2009/08/06 씹어먹는 C 언어 - <8. 우분투 리눅스에서 C 프로그래밍 하기>
11. 2009/08/15 씹어먹는 C 언어 - <9. 만약에... 2탄 (switch 문)>
12. 2009/08/15 씹어먹는 C 언어 - <10. 연예인 캐스팅(?) (C 언어에서의 형 변환)>
13. 2009/11/14 씹어먹는 C 언어 - <11 - 1. C 언어의 아파트 (배열), 상수>
14. 2009/10/29 씹어먹는 C 언어 - <11 - 2. C 언어의 아파트2 (고차원의 배열)>
15. 2009/11/09 씹어먹는 C 언어 - <12 - 1. 포인터는 영희이다! (포인터)>
16. 2009/11/14 씹어먹는 C 언어 - <12 - 2. 포인터는 영희이다! (포인터)>
17. 2009/11/26 씹어먹는 C 언어 - <12 - 3. 포인터는 영희이다! (포인터)>
18. 2009/12/14 씹어먹는 C 언어 - <13 - 1. 마술 상자 함수(function)>
19. 2009/12/19 씹어먹는 C 언어 - <13 - 2. 마술 상자 함수 2 (function)>
20. 2009/12/22 씹어먹는 C 언어 - <13 - 3. 마술 상자 함수 3 (function)>
21. 2009/12/27 씹어먹는 C 언어 - <13 - 4. 마술 상자 함수 (생각해볼 문제에 대한 아이디어)>
22. 2009/12/29 씹어먹는 C 언어 - <14. 컴퓨터의 머리로 따라가보자 - 디버깅(debugging)>
23. 2009/12/29 씹어먹는 C 언어 - <15 - 1. 일로와봐, 문자열(string)>
24. 2010/01/25 씹어먹는 C 언어 - <15 - 2. 일로와봐, 문자열(string) - 버퍼에 관한 이해>
25. 2010/02/01 씹어먹는 C 언어 - <15 - 3. 일로와봐, 문자열(string) - 문자열 지지고 볶기/리터럴>
26. 2010/02/08 씹어먹는 C 언어 - <15 - 4. 일로와봐, 문자열(string) - 도서 관리 프로젝트>
27. 2010/02/14 씹어먹는 C 언어 - <16 - 1. 모아 모아 구조체(struct)>
28. 2010/04/11 씹어먹는 C 언어 - <16 - 2. 모아 모아 구조체(struct) - 구조체 인자로 가진 함수>
29. 2010/06/13 씹어먹는 C 언어 - <16 - 3. 구조체와 친구들(공용체(union), 열거형(enum))>
30. 2010/06/19 씹어먹는 C 언어 - <17. 변수의 생존 조건 및 데이터 세그먼트의 구조>
31. 2010/07/16 씹어먹는 C 언어 - <18 - 1. 파일 뽀개기 (헤더파일과 #include) >
32. 2010/07/20 씹어먹는 C 언어 - <18 - 2. 파일 뽀개기 (# 친구들, 라이브러리)>

33. 2010/08/02 씹어먹는 C 언어 - <19. main 함수의 인자, 텅 빈 void 형>
34. 2010/08/03 씹어먹는 C 언어 - <20 - 1. 동동동 메모리 동적할당(Dynamic Memory Allocation)>
35. 2010/09/13 씹어먹는 C 언어 - <20 - 2. 메모리 동적할당 + 메모리 갖고 놀기>
36. 2010/11/21 씹어먹는 C 언어 - <21. 매크로 함수, 인라인 함수>
37. 2010/12/25 씹어먹는 C 언어 - <22. C 언어의 잡다한 키워드들 (typedef, volatile, #pragma)>
38. 2010/12/28 씹어먹는 C 언어 - <23 - 1. 파일 하고 이야기 하기 (파일 입출력의 기본적 이해)>
39. 2011/01/10 씹어먹는 C 언어 - <23 - 2. 파일 하고 이야기 하기 (파일 입출력)>
40. 2011/01/17 씹어먹는 C 언어 - <23 - 3. 파일 하고 이야기 하기 (파일 입출력 - 마무리)>
41. 2011/01/18 씹어먹는 C 언어 - <24. 더 빠르게 실행되는 코드를 위하여 (C 코드 최적화)>

위 많은 강좌들 중에서 여러분들이 한 번 더 보면 좋을 것이라 생각되는 강좌들은

- 2009/08/15 씹어먹는 C 언어 - <10. 연예인 캐스팅(?) (C 언어에서의 형 변환)>
- 2009/12/29 씹어먹는 C 언어 - <14. 컴퓨터의 머리로 따라가보자 - 디버깅(debugging)>
- 2010/01/25 씹어먹는 C 언어 - <15 - 2. 일로와봐, 문자열(string) - 버퍼에 관한 이해>
- 2010/02/01 씹어먹는 C 언어 - <15 - 3. 일로와봐, 문자열(string) - 문자열 지지고 볶기/리터럴>
- 2010/06/19 씹어먹는 C 언어 - <17. 변수의 생존 조건 및 데이터 세그먼트의 구조>
- 2010/08/03 씹어먹는 C 언어 - <20 - 1. 동동동 메모리 동적할당(Dynamic Memory Allocation)>
- 2010/12/25 씹어먹는 C 언어 - <22. C 언어의 잡다한 키워드들 (typedef, volatile, #pragma)>
- 2011/01/10 씹어먹는 C 언어 - <23 - 2. 파일 하고 이야기 하기 (파일 입출력)>
- 2011/01/18 씹어먹는 C 언어 - <24. 더 빠르게 실행되는 코드를 위하여 (C 코드 최적화)>

이 9 개의 강좌들 입니다. 왜냐하면 위 9 개의 강좌들에서는 여러분이 다른 곳에서 접하기 쉽지 않은 내용들이 많이 들어가 있기 때문에 제 강좌를 졸업 하기 전에 다시 한 번 읽어 두는 것이 많이 도움이 될 듯 합니다. 또한 C 언어를 보다 잘하기 위해서는 여러가지 표준 라이브러리 함수들과 친해지는 것이 중요한데, 제 블로그의 오른쪽 카테고리에 보면 C 언어 레퍼런스 라는 부분이 있습니다. 각 카테고리에 들어가셔서 나오는 여러가지 함수들과 친숙해 지는 것도 좋을 법 합니다.

자 그럼 위 강좌들도 다 읽고 C 언어로 왠만한 프로그램은 다 만들 수 있게 되었다면 어떻게 할까요. 그럼 이제 여러분들에게는 엄청나게 많은 선택권이 주어집니다. 일단 많은 사람들의 경우 다른 언어를 한 가지씩 더 배우게 됩니다. 대부분 C++ 아니면 Java 와 같은 객체 지향 언어(Object Oriented Programming Language) 를 배우거나 파이썬(Python) 과 같은 인터프리팅 형식의 언어들을 배울 수 도 있습니다.

저는 개인적으로 C++ 을 먼저 배우기를 추천합니다 :)

최근 들어서 아이폰 개발이나 안드로이드 개발에 붐이 일면서 Objective-C (아이폰), Java(안드로이드) 언어들을 배우는 사람들도 늘고 있습니다. 물론 이들을 배우는 것도 좋지만 개인적으로 생각해 볼 때 아무래도 C 를 배웠다면 'C 의 거의 대부분의 문법 요소들을 받아들인' C++ 을 배우는 것이 더 좋을 것 같네요.

그래서 저는 씹어먹는 C 언어를 끝내고 '씹어먹는 C++' 을 연재할 생각입니다. 더 다른 강좌 이름이 생각나신다면 댓글로 달아주시고요 ㅎㅎ 여러분들도 C 언어를 다 배웠다고 제 사이트를 잊지 말고 저의 두 번째 작품인 씹어먹는 C++ 에 관심을 가져 주시기 바랍니다.

저의 씹어먹는 C 언어 강좌를 TEX 형식 파일로 만들어주실 분 안계시나요 ??

씹어먹는 C 언어의 PDF 버전이 드디어 나왔습니다. 이제 모바일에서도 편하게 보실 수 있습니다. 여기를 클릭해서 확인하세요!